PROGRAMMEREN IN PROLOG

henk schotel

COUTINHO

programmeren in prolog

Omslag: Marijke Faber
Uitgever: Dick Coutinho, postbus 10, 1399 ZG Muiderberg

Voorwoord

Prolog is een programmeertaal die uitstekend geschikt is voor problemen, waarbij de te manipuleren gegevens complex gestructureerd zijn. Om die reden wordt Prolog hoe langer hoe meer gebruikt in het vaag afgegrensde gebied van de artificiële intelligentie, waarvan op dit moment expertsystemen de populairste toepassing zijn. Maar ook voor meer geformaliseerde vormen van symboolmanipulatie, zoals algebra, compilerbouw en computationele taalkunde is Prolog bij uitstek geschikt. Een programmeertaal die voor al deze gecompliceerde zaken geschikt is, kan uiteraard ook gebruikt worden voor meer prozaïsche programmatuur (databasesystemen, spreadsheets e.d.), zeker met de nu op de markt zijnde compilers en interpreters. Door zijn declaratieve karakter wijkt Prolog op een prettige manier af van imperatieve talen als Pascal en functionele talen als Lisp. Met Prolog kunnen al heel snel resultaten bereikt worden die in de meer conventionele talen een enorme programmeer - inspanning zouden vergen. Prolog is een *must* voor iedereen die wat programmeertalen betreft bij wil blijven. Maar aan de andere kant is Prolog ook uitstekend geschikt als eerste programmeertaal. Door bestudering van dit boek, maar vooral door het maken van de ruim 70 opgaven, kan een ieder zich een grondige kennis van de algemeen als standaard beschouwde versie van Prolog (Edinburgh Prolog) eigen maken.

Voor hun commentaar op eerdere versies van dit leerboek dank ik Dik Bakker (Alfa-Informatica, Universiteit van Amsterdam), Walter Daelemans (AI-lab, Vrije Universiteit Brussel), Fieny Pijls (Psychologisch Laboratorium, KU Nijmegen) en de volgende studenten van het eerste uur: Cor Baars, Philip Brey, Peter Elbert, André Koehorst en Willy Wolfs.
Voor cosmetische bijdragen ben ik in de eerste plaats dank verschuldigd aan Henk Maier (*terima kasih dan mohon maaf, ya Henk*) voor het langer dan gepland uitlenen van zijn computer met grafisch geörienteerd bedrijfssysteem. Verder aan Rob van de Sandt en Nettie Theyse voor hun hulp bij de eerste schreden op dit pas na enige tijd gebruikersvriendelijke apparaat. Ook dank ik de heer Eelants, die mij toegang heeft gegeven tot de laserprintfaciliteiten van het Psychologisch Laboratorium van de KUN.
Tenslotte dank ik de Faculteit der Wijsbegeerte van de KUN en in het bijzonder Prof. Dr. P.A.M. Seuren, die mij alle ruimte gegeven heeft dit boek te realiseren.

Voor commentaar houdt de auteur zich aanbevolen.

KUN-Faculteit der Wijsbegeerte
Postbus 980
6500 HK, Nijmegen.
(tel. 080-512949)

Inhoud

0 Proloog

0.1 Prolog en andere programmeertalen

Programmeertalen zijn ontworpen om problemen te beschrijven in een vorm die de oplossing van die problemen door een computer mogelijk maakt.
Eén zo'n programmeertaal is Prolog (**Pro**grammeren in **log**ica). Prolog is ontstaan door ideeën uit de predikatenlogica te combineren met de principes waarop de meer conventionele programmeertalen gebaseerd zijn. Hierdoor heeft Prolog een geheel eigen karakter.

0.1.1 Categorieën van programmeertalen

Programmeertalen kunnen op grond van het principe waarop ze gebaseerd zijn in drie categorieën ingedeeld worden: imperatieve, functionele en logische.
Een *imperatieve programmeertaal* is een taal waarbij de oplossing voor een probleem vertaald wordt in door de computer uit te voeren opdrachten. Complexen van bevelen kunnen gegroepeerd worden tot procedures die elkaar aanroepen. De meest gangbare programmeertalen (C, Pascal, Algol_68, Fortran, Cobol, Basic) behoren tot dit type.
Een *functionele programmeertaal* is een taal waarbij de oplossing van een probleem vertaald wordt in een geheel van elkaar aanroepende functies. Een functie wordt toegepast op een aantal objecten en het resultaat is een ander object. De meest gebruikte functionele taal op dit moment is LISP. Geëxperimenteerd wordt met FP, SASL, Hope en Miranda.
Een *logische programmeertaal* is een taal waarbij de logische restricties waaraan de oplossing van een probleem moet voldoen samenvallen met het computerprogramma dat dat probleem oplost. De eerste poging tot een logische programmeertaal is Prolog.
Functionele en logische programmeertalen hebben gemeen, dat ze beide een statische beschrijving van de oplossing van een probleem geven. Vandaar dat ze gezamenlijk wel aangeduid worden als de klasse van *declaratieve programmeertalen.*
Op dit moment bestaat er geen praktisch toepasbare programmeertaal die volledig in slechts een van de drie genoemde categorieën valt; zo kun je in de taal C functies definiëren; in de in de praktijk gebruikte dialecten van LISP komen functies voor met een imperatief karakter; Prolog tenslotte is weliswaar gebaseerd op de predikatenlogica, maar alleen door toevoeging van allerlei extralogische aspecten (zoals rekenfuncties) is deze taal een bruikbare programmeertaal geworden.

0.1.2 Artificiële Intelligentie en programmeertalen

Er bestaat een verband tussen de categorie waartoe een programmeertaal behoort en de aard van de problemen waarvoor die programmeertaal geschikt is. Functionele en logische programmeertalen zijn veel meer dan imperatieve talen geschikt voor *symbolisch programmeren,* dat wil zeggen voor het oplossen van problemen waarbij complexe, niet-numerieke gegevensstructuren een rol spelen. Dit verklaart de populariteit van LISP en Prolog in de Artificiële Intelligentie (AI), het veld van onderzoek dat probeert cognitieve vaardigheden van mens en dier op een computer na te bootsen. Cognitieve vaardigheden zijn onder meer lezen, spreken, schilderen, redeneren, ontwerpen, geloven, programmeren, leren en

televisiekijken. Bij al deze vaardigheden speelt kennis een centrale rol. De structuur van kennis en de functie ervan bij cognitieve processen zijn centrale thema's in het onderzoek van filosofen, psychologen, linguïsten en informatici. Dit onderzoek wordt gestimuleerd door het ontwikkelen van computermodellen. Ook praktische toepassingen, zoals expertsystemen, onderwijssystemen, relationele databases, auteursystemen en interfaces in natuurlijke taal, zijn eenvoudiger te realiseren in Prolog of LISP dan in een imperatieve taal.

0.1.3 Prolog en/of LISP

LISP is van oudsher de taal voor de AI. Prolog wint echter snel terrein, daar bij velen de indruk bestaat dat Prolog handiger is dan LISP voor symbolisch programmeren. Programma's in Prolog zijn over het algemeen beknopter dan programma's in LISP. Omdat zowel LISP als Prolog hun specifieke voor- en nadelen hebben, wordt op dit moment veel onderzoek gedaan naar mogelijkheden om functioneel en logisch programmeren op een goede manier te combineren.

0.1.4 Logisch programmeren en Prolog

Ondanks het feit dat de indeling in categorieën niet strikt is, verschillen de talen in de drie categorieën toch sterk van karakter. Vooral Prolog wijkt aanzienlijk af van de talen in de beide andere categorieën. De oorzaak hiervan is dat er bij het opstellen van een programma in Prolog mag worden uitgegaan van een mechanisme dat de dynamische samenhang (besturing) van de onderdelen van het programma regelt. Hierdoor zijn expliciete besturingsopdrachten (zoals *do while, if then* en *go to*) minder nodig dan in de overige talen. Kowalski, een van de grondleggers van het logisch programmeren, zegt het met een vergelijking:

Algoritme = Logica + Sturing

Een *algoritme* is de beschrijving van de stappen die doorlopen moeten worden om de oplossing van een probleem te vinden. De volgorde waarin de stappen moeten worden uitgevoerd wordt bepaald door de *logica* van die oplossing; voor de correcte volgorde van de stappen is *sturing* nodig. In een logische programmeertaal krijg je de sturing cadeau, dus hoeft er van een probleem alleen de logische structuur te worden beschreven. Maar zoals al gezegd is, Prolog is geen zuiver logische programmeertaal. In veel Prologprogramma's moet namelijk de gegeven besturingsfaciliteit expliciet beïnvloed worden, dus geldt voor Prolog de vergelijking:

Prologprogramma = Algoritme - deel van sturing

In wat volgt zullen we uiteenzetten hoe Prologprogramma's geconstrueerd kunnen worden en wat de impliciete besturingsfaciliteit van Prolog behelst. De begrippen *logische afleiding* (ook wel *logische inferentie* genoemd), *backtracking* ("terugkrabbelen") en *unificatie* zullen hierbij centraal staan.

0.2 Van predikatenlogica naar Prolog

Een *propositie* is een uitspraak die of *waar* is, of *onwaar*. De *predikatenlogica* is een formeel systeem waarmee men kan onderzoeken of een propositie *logisch afleidbaar* is uit andere, ware, proposities. In Prolog zijn alle afleidingen terug te brengen tot afleidingen van het type *Modus Ponens*. Een voorbeeld hiervan is:

Socrates *is* een mens.	(1) Gegeven propositie
Als X een mens is, dan is X sterfelijk.	(2) Gegeven implicatie
---	
Socrates is sterfelijk.	(3) Conclusie

We gaan deze logische afleiding vertalen in Prolog.
Daarvoor vertalen we de afleiding eerst in het formalisme van de predikatenlogica. In de predikatenlogica worden proposities ontleed in een *predikaat met een of meer argumenten* . Een predikaat zegt iets over een object (bijvoorbeeld "mens zijn"), of over de relatie tussen twee of meer objecten (bijvoorbeeld "broer van elkaar zijn"). In de predikatenlogica noteren we een propositie als een predikaat met daarachter de objecten als argumenten tussen haakjes:

predikaat(object_1, object_2, ...)

In (1) bijvoorbeeld zien we een *object* (Socrates) en een *predikaat* (mens). Dit noteren we als[1]:

mens(socrates). (1')

(2) is opgebouwd uit twee proposities met elk één argument, namelijk **mens(X)** en **sterfelijk(X)**, waarbij X een *variabele* is, die vervangen kan worden door een willekeurig object. Tussen de twee proposities bestaat een *als-dan* relatie. In de predikatenlogica heet deze relatie *implicatie* en het symbool daarvoor is een pijl: '->'
In predikatenlogica vertaald wordt (2):

mens(X) -> sterfelijk(X). (2')

De implicatie is een van de mogelijke relaties waarmee twee proposities kunnen worden gecombineerd tot meer complexe formules. De belangrijkste andere relaties die we ook in Prolog terugvinden zijn de *conjunctie* (*and*), de *disjunctie* (*or*) en de *negatie* (*not*). Deze zullen later, bij de bespreking van Prolog, ter sprake komen. In het formalisme van de predikatenlogica ziet onze afleiding er als volgt uit:

mens(socrates).	(1')
mens(X) -> sterfelijk(X).	(2')
---	
sterfelijk(socrates).	(3')

De omzetting naar Prolog is nu erg eenvoudig: (1') en (3') zijn al correct Prolog. Implicaties zoals (2') worden in Prolog omgekeerd: De proposities worden van plaats verwisseld en de pijl wordt vervangen door een *indien symbool*: ':-'. In Prolog wordt (2') dus:

sterfelijk(X) :- mens(X). (2'')

hetgeen gelezen moet worden als:

"X is sterfelijk INDIEN X een mens is".

Uiteindelijk ziet onze afleiding in Prolog er als volgt uit:

[1] *Alle (als) Prolog (bedoelde) tekst is vet afgedrukt.*

```
mens(socrates).                         (1")
sterfelijk(X) :- mens(X).               (2")
-----------------------------------------------------
sterfelijk(socrates).                   (3")
```

(1"), (2") en (3") heten in Prolog *Horn-clausules* (genoemd naar de logicus Horn). Een *Horn-clausule* bestaat uit een *propositie* (zoals 1"), of uit een *indien-relatie* (zoals 2"), met links van het indien-symbool (:-) één propositie en rechts daarvan óf een enkele propositie (zoals in 2") óf een uit meerdere proposities *samengestelde logische formule*. Een clausule zoals (1"), die uit een elementaire propositie bestaat wordt een *feit* genoemd; een formule zoals (2") waarin het 'indien symbool' voorkomt wordt een *regel* genoemd. Een *Prologprogramma* bestaat uit een geordende reeks Horn-clausules. Dus (1") en (2") vormen tezamen een Prologprogramma. We hadden (3") ook aan dit programmaatje kunnen toevoegen, maar dat is overbodig, omdat (3") afleidbaar is uit (1") en (2").
We zullen in dit boek verder geen bruggen slaan van de formele logica naar Prolog (zie daarvoor Lloyd, 1984), maar ons bezighouden met ons eigenlijke onderwerp: het opstellen van algoritmen in Prolog.

1 Het inferentiemechanisme, deel I

1.1 De Prologinterpretator en de Gegevensbank

Een Prologprogramma wordt verwerkt door een *Prologinterpretator*. Deze interpretator beschikt over een ***gegevensbank*** waarin clausules geplaatst kunnen worden. Om de afleidbaarheid van een propositie uit andere proposities te onderzoeken, moeten de gegeven clausules zich bevinden in de gegevensbank van de Prologinterpretator. Willen we bijvoorbeeld weten of

```
sterfelijk(socrates).                    (3")
```

afleidbaar is uit

```
mens(socrates).                          (1")
sterfelijk(X) :- mens(X).                (2")
```

dan moeten de twee laatste clausules in de gegevensbank staan en moeten we de interpretator opdracht geven te onderzoeken of (3") afleidbaar is uit de proposities die zich op dat moment in de gegevensbank bevinden. De propositie waarvan de interpretator de afleidbaarheid onderzoekt, heet de ***doelpropositie***. We zullen eerst beschrijven welke doelproposities mogelijk zijn, daarna hoe de interpretator te werk gaat bij het bepalen van de afleidbaarheid.

1.2 De doelpropositie

Er zijn twee typen doelproposities: doelproposities waarvan alle argumenten bekend zijn en doelproposities waarbij dat niet het geval is.

1.2.1 Doelproposities zonder onbekenden

Een voorbeeld van een doelpropositie waarin alle argumenten bekend zijn, is:

```
sterfelijk(socrates).                    (3")
```

Andere voorbeelden zijn:

```
mens(socrates).
gifbeker(socrates,-399).
```

Ook doelproposities *zonder argumenten* behoren tot dit type, zoals:

```
doe_maar.
```

De interpretator zal bij een doelpropositie zonder onbekenden uitsluitend onderzoeken of de

aangeboden propositie afleidbaar is. Anders is dat wanneer er variabelen in de doelpropositie voorkomen.

1.2.2 Doelproposities met variabelen, non-determinisme

Een doelpropositie kan ook argumenten bevatten waarvan de waarde onbekend is. Dergelijke argumenten worden aangegeven door middel van een *logische variabele.* Of een argument variabel is of niet, zien wij (en ook de Prologinterpretator!) aan het eerste teken waarmee dat argument wordt aangeduid: is dat een *hoofdletter* of een *laagliggend streepje* (_), dan is het argument een logische variabele. Voorbeelden van variabelen zien we in onderstaande proposities:

```
mens(X).
sterfelijk(Wie).
broer(jan,_broer).
broer(Persoon_1,Persoon_2).
predikaat(VAR,en,VAR,is_gelijk,VAR).
predikaat(Var,en,Var,is_gelijk,VAR).
```

X, Wie, _broer, Persoon_1, Persoon_2, VAR en *Var* zijn *variabelen; jan, en* en *is_gelijk* daarentegen zijn *geen* variabelen. Het zal nu ook duidelijk zijn waarom we in proposities zo oneerbiedig zijn geweest Socrates' naam te spellen zonder hoofdletter!

Een doelpropositie met variabelen wordt door de interpretator opgevat als een opdracht om in de gegevensbank te zoeken naar waarden die de doelpropositie afleidbaar maken. De doelproposities hierboven kunnen dan ook geïnterpreteerd worden als:

Zoek een waarde voor *X*, zodanig dat **mens(X)** afleidbaar is.
Zoek een waarde voor *Wie*, zodanig dat **sterfelijk(Wie)** afleidbaar is.
Zoek een combinatie van waarden voor *Persoon_1* en voor *Persoon_2*,
    zodanig dat **broer(Persoon_1,Persoon_2)** afleidbaar is.
Zoek een waarde voor *VAR*,
    zodanig dat **predikaat(VAR,en,VAR,is_gelijk,VAR)** afleidbaar is.
Zoek een combinatie van waarden voor *Var* en *VAR*,
    zodanig dat **predikaat(Var,en,Var,is_gelijk,VAR)** afleidbaar is.

Wordt er een combinatie van waarden voor de variabelen gevonden die de doelpropositie afleidbaar maakt, dan worden deze waarden aan de variabelen toegekend. Anders uitgedrukt: *de* variabelen worden *gebonden* aan de gevonden waarden. Zo'n *binding van variabelen* wordt een *oplossing* of een *substitutie* genoemd. Prolog is *non-deterministisch,* wat wil zeggen dat de Prologinterpretator, indien dat gewenst is, verscheidene oplossingen kan geven voor de variabelen in een doelpropositie. Vaak is er inderdaad meer dan één substitutie mogelijk. Bijvoorbeeld: laat de gegevensbank bevatten:

```
mens(socrates).              (1")
mens(russell).               (4")
sterfelijk(X) :- mens(X).    (2")
```

dan is de doelpropositie **mens(Wie)** afleidbaar als de variabele *Wie* gelijk is aan *socrates,*

Wie = socrates

maar ook als *Wie* gelijk is aan *russell:*

Wie = russell

De doelpropositie **mens(Wie)** heeft dus twee oplossingen.

Soms wordt ook helemaal geen oplossing gevonden: de doelpropositie **dier(D)** levert bij bovenstaande gegevensbank geen enkele oplossing voor de variabele ***D*** op, omdat er voor deze variabele in de gegevensbank geen enkele waarde te vinden is die die propositie afleidbaar maakt (het predikaat ***dier*** is in onze gegevensbank helemaal niet gedefinieerd).

1.3 *Yes* en *No*, TRUE en FAIL

Het resultaat van een onderzoek naar de afleidbaarheid van een doelpropositie leidt tot één van de twee mogelijke antwoorden van de interpretator: *Yes* of *No*.
Yes betekent dat de afleidingspoging succesvol was; de doelpropositie was "waar", **true**. *No* betekent het tegendeel. De term voor het mislukken van een afleiding is **fail.**
Indien er in een doelpropositie variabelen voorkomen en de propositie afleidbaar is, dan zal de interpretator, voordat het antwoord *Yes* gegeven wordt, eerst één of meer oplossingen aan de programmeur rapporteren.

1.4 Unificatie (versie 1)

Fundamenteel in het afleidingsproces is de *unificatie van de doelpropositie met een propositie uit de gegevensbank.* Twee proposities zijn *unificeerbaar* indien ze door middel van een substitutie aan elkaar gelijk kunnen worden gemaakt. Dit is het geval indien:

1. de proposities hetzelfde predikaat hebben (predikaten zijn niet variabel);
2. de proposities hetzelfde aantal argumenten hebben;
3. *alle* qua plaats corresponderende argumenten met elkaar unificeerbaar zijn.

Er zijn drie gevallen waarin twee corresponderende argumenten unificeerbaar zijn.

a. Beide argumenten zijn constanten en gelijk aan elkaar.
b. Het ene argument is een variabele, het andere niet.
c. Beide argumenten zijn variabelen.

Zijn twee proposities unificeerbaar, dan vindt de *unificatie* van beide proposities plaats door aan de variabelen in beide proposities de corresponderende elementen als waarde toe te kennen.
Voorbeelden:

Propositie 1	*Propositie 2*	*Substitutie*
mens(socrates)	**filosoof(socrates)**	niet unificeerbaar
mens(socrates)	**mens(plato)**	niet unificeerbaar
mens(socrates)	**mens**	niet unificeerbaar!
mens	**mens**	geen, wél unificeerbaar!
mens(socrates)	**mens(socrates)**	geen, wél unificeerbaar!
mens(X)	**mens(socrates)**	X = socrates
mens(X)	**mens(Mens)**	X = Mens, twee variabelen!
mens(X)	**mens(X)**	X = X

Propositie 1	*Propositie 2*	*Substitutie*
vader(V,Z)	**vader(jan,piet)**	V = jan, Z = piet
vader(V,piet)	**vader(jan,Z)**	V = jan, Z = piet
vader(jan,Z)	**vader(V,piet)**	V = jan, Z = piet
vader(jan,Z)	**vader(V,Z)**	V = jan, Z = Z
vader(V,Z)	**vader(jan,jan)**	V = jan, Z = jan
vader(V,jan)	**vader(jan,Z)**	V = jan, Z = jan
vader(jan,Z)	**vader(V,jan)**	V = jan, Z = jan
vader(X,X)	**vader(jan,jan)**	X = jan
vader(X,jan)	**vader(jan,X)**	X = jan
vader(jan,X)	**vader(X,jan)**	X = jan
vader(X,X)	**vader(jan,piet)**	niet unificeerbaar
vader(X,piet)	**vader(jan,X)**	niet unificeerbaar
vader(jan,X)	**vader(X,piet)**	niet unificeerbaar

De paren van proposities in de drie laatste voorbeelden zijn niet unificeerbaar, omdat een variabele in een substitutie niet twee waarden tegelijk aan kan nemen: ***X*** kan niet tegelijkertijd gelijk zijn aan ***jan*** en aan ***piet***.

1.5 Bepaling van de afleidbaarheid

Hoe bepaalt de interpretator of de doelpropositie afleidbaar is uit de clausules in de gegevensbank? Of anders gezegd: hoe werkt het *inferentiemechanisme* van de Prolog-interpretator? In dit hoofdstuk zullen we deze vraag beantwoorden voor de eenvoudigste gevallen, met name *backtracking* zal nog wat onderbelicht worden. Dat zal worden goedgemaakt in hoofdstuk 3.

De propositie links van het indien-teken in een regel heet het *hoofd* van die regel. Voor het bepalen van de afleidbaarheid van een doelpropositie onderzoekt de interpretator of zich in de gegevensbank een feit of hoofd van een regel bevindt waarmee de doelpropositie geünificeerd kan worden.Wordt een *feit* gevonden, dan is de afleiding direct geslaagd; wordt een regel gevonden, dan volgt een *reductiestap.* Dit houdt in dat de interpretator onderzoekt of het rechterdeel van de regel afleidbaar is. Is dat het geval, dan is onze oorspronkelijke doel - propositie afleidbaar. Een afleiding is meer of minder gecompliceerd, afhankelijk van het aantal regels dat daarbij een rol speelt. We beginnen met het bespreken van afleidingen waarin helemaal geen regels toegepast worden, maar waarbij één unificatiestap voldoende is. Daarna volgen afleidingen waarin regels toegepast worden.

1.5.1 Afleidingen bestaande uit één unificatiestap

1.5.1.1 Doelproposities zonder variabelen

Een doelpropositie zonder variabelen is afleidbaar in één *unificatiestap* wanneer in de gegevensbank een feit staat dat unificeerbaar is met de doelpropositie.
Als we bijvoorbeeld uitgaan van de volgende gegevensbank,

```
mens(socrates).                    (1")
mens(russell).                     (4")
sterfelijk(X) :- mens(X).          (2")
```

en we hebben als doelpropositie

```
mens(socrates).                    (3")
```

dan leidt dit tot het antwoord *yes* van de interpretator, omdat de doelpropositie gelijk is aan (1") in de gegevensbank. De doelpropositie

```
mens(piet).
```

daarentegen resulteert in het antwoord *no* van de interpretator. Ook een propositie met een predikaat anders dan ***mens*** of ***sterfelijk*** (de twee enige predikaten in ons programma) leidt tot een negatief resultaat, bijvoorbeeld

```
filosoof(socrates).
broer(jan,piet).
```

leiden beide tot het antwoord *no*.

De doelpropositie **mens(socrates)** is ook unificeerbaar met een propositie met een variabele als argument, bijvoorbeeld met:

```
mens(X).
```

Zulke *algemene feiten* zijn alleen in heel bepaalde gevallen nuttig in de gegevensbank, maar soms staan ze er per ongeluk in, wat tot ongewenst positieve resultaten kan leiden.

1.5.1.2 Doelproposities met variabelen

Ook een doelpropositie met variabelen is afleidbaar in één *unificatiestap* wanneer in de gegevensbank een feit staat dat unificeerbaar is met de doelpropositie. Wordt zo'n feit gevonden, dan worden de waarden die aan de variabelen van de doelpropositie werden toegekend door de interpretator gerapporteerd.Als we weer uitgaan van de volgende gegevensbank

```
mens(socrates).                    (1")
mens(russell).                     (4")
sterfelijk(X) :- mens(X).          (2")
```

en we hebben als doelpropositie

```
mens(X).
```

dan zijn er twee oplossingen mogelijk: X = socrates en X = russell.

Of beide oplossingen op het scherm verschijnen hangt af van degene die de doelpropositie aan de interpretator aangeboden heeft. Wanneer de interpretator een oplossing gevonden heeft, wordt deze op het scherm getoond en wordt er gewacht op de reactie van de gebruiker. Als er een *puntkomma* wordt ingetypt, wordt de laatste elementaire afleiding teruggenomen. De

daarbij horende bindingen van variabelen worden *ongedaan* gemaakt, waarna naar de volgende oplossing gezocht wordt. Met andere woorden, de interpretator krabbelt een beetje terug en gaat dan langs een andere weg weer vooruit. *De volgorde waarin de oplossingen gevonden worden, is gelijk aan de volgorde waarin de proposities in de gegevensbank staan* . Als na een gevonden oplossing de ***return*** (of *enter*)toets wordt ingedrukt, worden er geen andere oplossingen gezocht, maar verschijnt *no* op het scherm[i] .

Bij de behandeling van de doelpropositie uit ons voorbeeld

mens(X).

gaat de Prologinterpretator op zoek naar een propositie in de gegevensbank waarvan het predikaat ***mens*** is en het aantal argumenten gelijk is aan 1. De eerste propositie in de gegevensbank die voldoet is **mens(socrates)**; ***socrates*** correspondeert met de variabele *X*, dus **mens(X)** is afleidbaar als *X* gelijkgesteld wordt aan ***socrates***. De Prologinterpretator toont dus op het scherm:

X = socrates

Het intypen van een puntkomma levert een tweede oplossing:

X = russell

Aangezien er maar twee oplossingen zijn, krijgen we na nog een puntkomma geen andere oplossing meer en verschijnt er *no* in beeld.

We hadden ook een andere, meer omschrijvende naam voor de variabele kunnen kiezen, zoals in:

mens(Mens).

Dit had als eerste oplossing opgeleverd:

Mens = socrates

Als laatste voorbeeld beschouwen we de doelpropositie

mens(x).

Deze levert geen enkele waarde op, aangezien er geen variabele in de vraag staat en de propositie **mens(x)** niet in de gegevensbank voorkomt. Het enige resultaat is dat de interpretator *no* antwoordt.

1.5.2 Toepassing van een regel (reductiestap)

Voor het bepalen van de afleidbaarheid van een doelpropositie zoekt, zoals reeds vermeld werd, de interpretator niet alleen naar feiten in de gegevensbank, maar ook naar regels waarvan de propositie links van het indien-teken unificeerbaar is met de doelpropositie. Wordt zo'n regel gevonden, dan is die regel *toepasbaar*. Toepassing van de regel houdt in dat de interpretator onderzoekt of het rechterdeel van de regel afleidbaar is. Is dat het geval, dan is onze

[i] *Dit is implementatieafhankelijk: soms betekent een return stoppen met zoeken en elk ander teken verder gaan met zoeken.*

oorspronkelijke doelpropositie afleidbaar. We zien dus dat een regel tot een andere doelpropositie kan leiden, welke afkomstig is van het rechterdeel van die regel. Elke geslaagde afleiding eindigt met een elementaire afleiding, een unificatiestap, zoals beschreven in de vorige sectie. Door toepassing van een regel wordt een probleem tot een of meer andere gereduceerd, vandaar dat de toepassing van een regel ook wel een *reductiestap* genoemd wordt.

1.5.2.1 Doelproposities zonder variabelen

Het hoofd van een toe te passen regel zal in vrijwel alle gevallen een variabele bevatten die correspondeert met een argument van de doelpropositie.
Gaan we weer uit van de gegevensbank bestaande uit

```
mens(socrates).                    (1")
mens(russell).                     (4")
sterfelijk(X) :- mens(X).          (2")
```

dan correspondeert ***socrates*** in de doelpropositie

```
sterfelijk(socrates).
```

met de ***X*** in

```
sterfelijk(X) :- mens(X).          (2")
```

Voordat de interpretator verder gaat met het onderzoeken van de afleidbaarheid van de propositie rechts van het indien-symbool, wordt aan de variabele uit het hoofd van de regel de constante uit de doelpropositie als waarde toegekend. Dus in ons voorbeeld unificeert de substitutie

X = socrates

de doelpropositie met het hoofd van regel (2"). Deze substitutie zal niet door de interpretator op het scherm getoond worden, omdat nu de variabele niet staat in de doelpropositie maar in een clausule in de gegevensbank. Na de substitutie treedt een uiterst belangrijk principe in werking: ***Een substitutie voor een variabele in een regel geldt voor die gehele regel,*** dus zowel voor de linker- als voor de rechterkant van die regel. Hierbij wordt de regel in de gegevensbank niet gewijzigd, maar de interpretator voegt een kopie van de regel toe aan de afleiding en registreert daarbij welke substitutie geldt.
Ten behoeve van de afleiding in ons voorbeeld wordt (2") door de interpretator dus gelezen als:

```
sterfelijk(socrates) :- mens(socrates). (2"')
```

De substitutie geldt alleen voor deze afleiding en slechts voor deze ene clausule maar niet voor andere clausules in de gegevensbank, ook al bevatten die variabelen met dezelfde naam. Kort samengevat, ***het bereik van een variabele in een clausule is die clausule.***
In ons voorbeeld leidt regel (2") via regel (2"') tot de volgende nieuwe doelpropositie:

```
mens(socrates).
```

Deze propositie is afleidbaar in een unificatiestap, omdat deze propositie letterlijk in de gegevensbank staat (1"). De Prologinterpretator laat daarom door middel van het antwoord *yes* weten dat onze oorspronkelijke doelpropositie (3") afleidbaar is uit de proposities in de gegevensbank.

1.5.2.2 Doelproposities met variabelen

Wanneer een variabele in de doelpropositie correspondeert met een variabele van een in de gegevensbank gevonden propositie (de namen van de variabelen hoeven hiervoor niet aan elkaar gelijk te zijn!), maakt de interpretator ook in dit geval een kopie van de toepasbare regel en onthoudt dat beide variabelen gelijke substituties moeten krijgen. Vervolgens wordt de rechterzijde van de regel op afleidbaarheid onderzocht, leidt dit tot een substitutie voor de variabele uit de regel, dan geldt de toegewezen waarde ook voor de variabele uit de doelpropositie. Slaagt de afleiding zonder dat er voor de variabele een waarde gevonden wordt, dan rapporteert de interpretator de gelijkstelling van twee variabelen. Meestal duidt dit op een onbedoeld algemeen feit in de gegevensbank (zie *1.5.1.1*). Evenals bij de elementaire afleidingen is ook hier meer dan een oplossing mogelijk.

De vraag naar alle sterfelijke objecten in de gegevensbank bijvoorbeeld, luidt als doelpropositie:

sterfelijk(Object).

Alleen regel (2") correspondeert met de doelpropositie. Met de variabele Object correspondeert de variabele X uit het hoofd van de regel. Dus we krijgen een substitutie waarin deze twee variabelen aan elkaar gelijkgesteld worden:

Object = X

Vervolgens wordt het rechterdeel van de regel op afleidbaarheid onderzocht, waardoor

mens(X).

de nieuwe doelpropositie is. Dit leidt, zoals we al bij de behandeling van de elementaire afleidingen gezien hebben, tot de waardetoekenning

X = socrates

en aangezien X en Object geünificeerd waren, geldt ook:

Object = socrates

Alleen deze laatste substitutie wordt door de interpretator op het scherm zichtbaar gemaakt, omdat ***Object*** als variabele in de doelpropositie aanwezig was. Vervolgens kunnen we door een puntkomma aangeven dat er nieuwe oplossingen moeten worden gezocht. De gevonden substitutie voor ***X***, en ook die voor ***Object*** worden opgeheven en er wordt een nieuwe oplossing voor mens(X) gezocht en gevonden, namelijk

X = russell

waarmee tevens gevonden wordt:

Object = russell

1.6 Opgaven

Het wordt na alle voorgaande beschouwingen de hoogste tijd om aan de computer te gaan zitten. Maar voordat we in het volgende hoofdstuk zien hoe de interactie met de machine verloopt, bieden de volgende opgaven de gelegenheid eerst nog wat droog te zwemmen.

We gaan uit van een gegevensbank die de volgende clausules bevat.

```
poes(tom).
poes(garfield).

hond(pluto).
hond(snoopy).
hond(tom).

miauwen(X) :- poes(X).
blaffen(X) :- hond(X).
```

1.1 Wat zal de interpretator antwoorden wanneer onderstaande proposities als doelpropositie worden aangeboden? Bevat een doelpropositie een variabele, geef dan aan wat er ingetypt moet worden om na een oplossing de volgende oplossing te krijgen en wat het resultaat is als er geen oplossing meer is.

(a) **poes(garfield).**
(b) **hond(garfield)**
(c) **poes(tom).**
(d) **hond(tom).**
(e) **hond(lassie).**
(f) **poes(X).**
(g) **kat(garfield).**
(h) **poes(Tom).**

1.2 Beantwoord dezelfde vragen voor:

(a) **miauwen(garfield).**
(b) **blaffen(garfield).**
(c) **miauwen(tom).**
(d) **blaffen(tom).**
(e) **blaffen(Poes).**
(f) **blaffen(lassie).**
(g) **miauwen(x).**
(h) **mauwen(X).**

2 Interactie met de computer

2.1 Programmeeromgevingen

Onder een *programmeeromgeving* verstaan we de hulpmiddelen die ons bij het ontwikkelen van een programma ter beschikking staan. Tot de programmeeromgeving behoren naast een computer en de *interpretator/vertaler* voor de gebruikte programmeertaal ook een *editor*, waarmee programma's in een *file* op een schijf kunnen worden gezet en gewijzigd. In de eenvoudigste, minst gebruikersvriendelijke programmeeromgeving voor Prolog op een *personal computer (PC)* zijn de interpretator en de editor volkomen onafhankelijk van elkaar zodat in een Prologsessie de volgende stappen doorlopen worden:

1. Zet de machine aan en start het bedrijfssysteem (bijv. MS-DOS).

2. *Edit*: Zet het programma in een (of meer) file(s) op schijf

3. Herhaal de stappen 3.1 en 3.2 totdat het programma in orde is:

    3.1. *Prolog*: Start de PROLOG interpretator, vervolgens:

    3.1.1. **consult** of **reconsult**:
    Vul de gegevensbank vanuit de programmafile(s).
    Lukt dit niet door syntactische fouten in het programma, ga dan door met 3.1.3 en 3.2.

    3.1.2. *Test*
    Onderzoek of het programma doet wat het moet doen door relevante doelproposities aan te bieden. Gedraagt het programma zich erg vreemd, gebruik dan *trace* en *spy* om de fouten op te sporen (zie volgende hoofdstuk).

    3.1.3. **halt**:
    Stop de interpretator (de gegevensbank houdt op te bestaan!) en ga terug naar het bedrijfssysteem.

    3.2. *Edit*: Verbeter de programmafile(s).

4. Zet de machine uit.

Er zijn gelukkig ook voor Prolog al programmeeromgevingen waarbij de editor en de interpretator goed geïntegreerd zijn, waardoor de programmaontwikkeling aanzienlijk sneller en prettiger gaat dan hierboven geschetst. In de meeste gevallen is het mogelijk om de editor te starten zonder de Prologinterpretator te stoppen. In een dergelijke omgeving zijn de stappen 3.2 en 3.1.3 dus verwisseld. Wel moet de editor hierbij elke keer opnieuw worden gestart en

gestopt. Er zijn implementaties van Prolog waarbij ook dit niet nodig is, bijvoorbeeld omdat de file-editor deel uitmaakt van de Prologinterpretator. Een andere factor die het karakter van de programmeeromgeving beïnvloedt is of de implementatie "window-oriented" is of niet[1]. Het zal niet zo lang meer duren of er bestaan snelle Prologwerkstations, waarbij de hele computer ingericht is voor de ontwikkeling en uitvoering van Prologprogramma's, net zo als er LISP-machines bestaan voor de ontwikkeling van LISP-programma's.
Aangezien er geen standaard programmeeromgeving bestaat voor Prolog zullen we in dit boek uitgaan van de meest simpele programmeeromgeving bestaande uit een PC zonder vaste schijf, met MS-DOS als bedrijfssysteem en een interpretator zonder editor. Voor het maken en wijzigen van programmafiles zij men verwezen naar de bij de computer en interpretator behorende manuals.

2.2 *Public Domain* software

Voor MS-DOS en ook voor andere systemen zijn "public domain" (gratis) Prologimplementaties voor beginners in omloop, zoals "Toy Prolog" van Kluzniak en Szpakowicz(1985). Ook EMACS, de meest gebruikte editor in AI kringen, is vrij verkrijgbaar bijvoorbeeld via het Fido netwerk (079-510425) van de Hobby Computer Club. Voor professionele implementaties van Prolog zij men verwezen naar de laatste bladzijden in dit boek.

2.3 Opstarten MS-DOS

Voordat we de machine en de monitor aanzetten, plaatsen we de MS-DOS systeemschijf in diskette station A (*drive A*). We zetten de machine aan en beantwoorden eventuele vragen van MS-DOS over de datum en de tijd. Kort daarna verschijnt 'A>' links op een regel die verder leeg is. Dit is de *prompt* van MS-DOS. De prompt van een systeem geeft aan dat het klaar staat om opdrachten te ontvangen en uit te voeren. Aan de prompt herkent men met welk systeem men aan het interacteren is.

2.4 Starten en vragen

Nadat we MS-DOS gestart hebben typen we achter de prompt de naam van de Prologinterpretator gevolgd door een *return* of *enter* (afhankelijk van het toetsenbord). In de meeste gevallen heet de Prologinterpretator *Prolog*, zodat we op het scherm zien:

```
A>prolog
```

Dit heeft tot gevolg dat MS-DOS de Prologinterpretator start en de controle overdraagt aan dit systeem, hetgeen blijkt uit een nieuwe prompt, die er bijvoorbeeld zo uit ziet:

```
| ?-
```

Vanaf nu "zitten we in Prolog". De prompt duidt aan dat de Prologinterpretator wacht op een opdracht die een afleiding op gang moet brengen. Zo'n opdracht noemen we een *vraag*. De eenvoudigste vraag heeft de vorm van een propositie: typen we achter de prompt een *propositie gevolgd door een punt* in, dan start de interpretator *na het aanslaan van de return* (of *enter*) *toets*

[1] *Er bestaat een systeem ("Jumbo Prolog" of zoiets) dat weliswaar een schitterende programmeeromgeving biedt, maar waarvan de programmeertaal te veel kenmerken van Pascal en te weinig kenmerken van Prolog bezit om als een implementatie van Prolog beschouwd te kunnen worden.*

een afleiding waarbij de propositie als doelpropositie fungeert. Het resultaat van de afleiding wordt op het scherm afgebeeld en de interpretator wacht op de volgende vraag.
Bijvoorbeeld: Hebben we een lege gegevensbank en willen we weten of er sterfelijke objecten zijn dan typen we **sterfelijk(X)** gevolgd door een punt en een *return* (of *enter*) en bekijken het antwoord (*no*). Op het scherm zien we:

```
| ?-sterfelijk(X).
no
```

Bevat onze gegevensbank de volgende clausules:

```
mens(socrates).                (1")
mens(russell).                 (4")
sterfelijk(X) :- mens(X).      (3")
```

dan is de volgende dialoog mogelijk (we stellen meerdere vragen):

```
| ?-mens(russel).
no
| ?-mens(russell).
yes
| ?-sterfelijk(socrates).
yes
| ?-sterfelijk(X).
X = socrates;
X = russell;
no
```

Een fout die in het begin veel gemaakt wordt is dat na de doelpropositie alleen een *return* wordt ingetypt. De punt wordt vergeten en men gaat op het antwoord zitten wachten. Vaak typt men nog wat *returns* in. Er komt geen antwoord, omdat de interpretator ook wacht en wel op een punt, gevolgd door een return. De remedie is de punt en de return alsnog in te typen, bijvoorbeeld:

```
| ?-sterfelijk(socrates)

.
yes
| ?-sterfelijk(X)



.
X = socrates;
X = russell
yes
```

(Bij de laatste vraag volgt de punt pas na twee returns.)

2.5 Stoppen van Prolog; systeempredikaten

Kennis van Prolog behelst naast het kennen van het inferentiemechanisme ook het kennen van enige tientallen voorgedefinieerde predikaten.

Bij een aantal hiervan gaat het niet zozeer om het op gang brengen van een afleiding, maar om de imperatieve neveneffecten. Een voorbeeld daarvan is het argumentloze predikaat **halt**. Wanneer we **halt** als doelpropositie aanbieden wordt de Prologinterpretator gestopt en wordt de controle weer overdragen aan het bedrijfssysteem. We zien dan weer de MS-DOS prompt. De kortst mogelijk Prologsessie is:

```
A> Prolog
| ?-halt.
A>
```

Zelfs een vriendelijk *Yes* kan er niet meer van af! De gegevensbank, die zich alleen in het interne werkgeheugen van de interpretator bevindt, bestaat nadat we Prolog verlaten hebben uiteraard ook niet meer. In de meeste gevallen zal **halt** niet de eerste vraag zijn die we het systeem laten beantwoorden. Toch hebben we **halt** als eerste systeempredikaat behandeld, omdat het erg frustrerend kan zijn niet te weten hoe je weer uit Prolog komt.
Ook het vullen van de gegevensbank en het bekijken van de clausules in de gegevensbank is mogelijk met behulp van systeempredikaten. Voordat deze besproken kunnen worden is het nodig te weten wat onder een *constante* verstaan wordt.

2.6 Constanten: atomen en getallen

Een propositie kan twee typen *constanten* bevatten: atomen en getallen.
Een *atoom* is een tekstuele constante die opgebouwd is uit letters, cijfers en speciale tekens. In bepaalde gevallen moet een atoom tussen twee *apostrophen* (') staan. De meeste atomen worden zo gekozen dat de apostrophen overbodig zijn. Dit is in de eerste plaats het geval wanneer een atoom aan de volgende eisen voldoet:

(1) het eerste teken is een kleine letter,
(2) het atoom bevat verder uitsluitend letters, cijfers of *laagliggende* streepjes.

Een tweede type atoom waarbij de apostrophen achterwege mogen blijven zijn de atomen die geheel uit speciale tekens bestaan (bijvoorbeeld: :- en -->). Hierbij kunnen problemen ontstaan wanneer een dergelijk atoom een speciale betekenis voor het systeem heeft, maar dit is afhankelijk van de gebruikte Prologimplementatie. In hoofdstuk 10 komen we hierop terug.

Voldoet een atoom niet aan een van de hiervoor beschreven criteria, dan moet het tussen apostrophen staan. In dat geval zijn de vreemdste constructies mogelijk. Ook apostrophen zijn dan toegestaan, mits een apostrophe die tot het atoom behoort met dubbele apostrophen wordt aangegeven. Zoals in ***'''s Avonds'***. De twee apostrophen (twee toetsaanslagen) moeten hierbij niet verward worden met een aanhalingstaken (één toetsaanslag).

Als, per ongeluk, in plaats van een laagliggend streepje een minteken gebruikt wordt in wat een atoom zonder omringende apostrophen zou moeten zijn, dan zal dat in veel gevallen geen probleem geven, ondanks het feit dat dan geen sprake is van een atoom, maar van een arithmetische *structuur* (zie hoofdstuk 7). Consequent gebruik van het laagliggende streepje is efficiënter en voorkomt fouten.

Over *getallen* valt weinig meer te vermelden dan dat ze uit cijfers bestaan, al of niet voorzien van een plus- of minteken. In hoofdstuk 7 komen we uitvoeriger over getallen te spreken. Let wel: ***-399*** is een *getal*, ***'-399'*** is een *atoom*.

Met behulp van het nieuwe begrip *atoom* kunnen we wat al eerder over predikaten werd opgemerkt als volgt uitdrukken: het predikaat van een propositie is altijd een atoom.

2.7 Kijken in de gegevensbank

Het predikaat **listing** zonder argumenten toont *alle* clausules die zich in de gegevensbank bevinden op het scherm. (De systeempredikaten bevinden zich niet in de gegevensbank, dus de definities daarvan worden niet getoond door **listing**) We kunnen ook selectief in de gegevensbank kijken door een predikaat als argument aan **listing** mee te geven: in dat geval worden alle feiten met dat predikaat en alle regels met een hoofd dat dat predikaat bevat op het scherm afgebeeld. Voorbeelden:

| **?-listing.**
mens(socrates).
mens(russell).
sterfelijk(eenhoorn).
sterfelijk(X) :- mens(X).
yes
| **?-listing(sterfelijk).**
sterfelijk(eenhoorn).
sterfelijk(X) :- mens(X).
yes
| **?-listing(socrates).**
yes

Het laatste voorbeeld toont niets op het scherm, omdat ***socrates*** geen predikaat is!
Merk op dat **listing** desondanks *yes* als resultaat van de afleiding oplevert. Er zijn meer systeempredikaten die altijd tot het antwoord *yes* leiden.

2.8 Vullen van de gegevensbank

Voor het vullen van de gegevensbank in het interne werkgeheugen staan de volgende systeempredikaten ter beschikking: **consult, reconsult, assert, asserta** en **assertz.**

2.8.1 consult

Een Prologproramma wordt bewaard in een of meer files op een schijf. Om door Prolog gebruikt te kunnen worden, moet een Prologprogramma zich bevinden in de gegevensbank in het snelle, interne geheugen van de computer. Prolog copieert een programma van een file naar het interne geheugen door middel van het **consult** commando. Hierbij worden de clausules *toegevoegd* aan de al in de gegevensbank aanwezige clausules. (**consult** is weer een voorbeeld van een predikaat waarbij het om de nevenwerking gaat en dat ook altijd *yes* oplevert als resultaat van de afleiding). Het **consult** predikaat verlangt als argument een *atoom* gelijk aan de naam van de te consulteren file. Laten we aannemen dat de clausules (1") en (2") uit het vorige hoofdstuk zich in een file genaamd *socrates.pro* bevinden. Om deze clausules "binnen te halen", moeten we de Prologinterpretator de volgende vraag stellen:

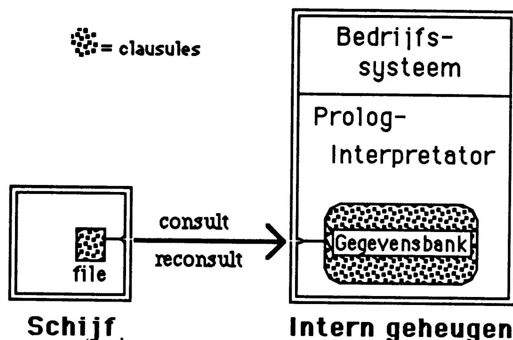
| **?-consult('socrates.pro').**
yes

Hierna kunnen we dan vragen zoals (3") intypen, die tot de eerder besproken resultaten leiden:

```
| ?-sterfelijk(socrates).
yes
| ?-sterfelijk(X).
X = socrates;
no
```

(Wat zou in bovenstaand voorbeeld het resultaat zijn wanneer (3") als doelpropositie aangeboden werd *voordat* het **consult** commando wordt uitgevoerd?)

In een schema ziet de relatie tussen files en de gegevensbank er als volgt uit:



Een speciale filenaam die het *toestenbord* aanduidt, is ***user*** (MS-DOS systemen gebruiken hiervoor ook wel ***con***, van *console*). Doen we:

```
| ?-consult(user).
```

dan verschijnt er (bij veel Prologimplementaties) een nieuwe *prompt*, bijvoorbeeld:

```
|
```

Het systeem wacht nu op clausules die *ingetikt moeten worden vanaf het toetsenbord.* Elke clausule moet worden afgesloten door een punt en een *return* (of *enter*). Als alle clausules zijn ingetypt, moet de file als geheel afgesloten worden. Dit geschiedt door het gelijktijdig indrukken van de *control* toets en de laatste letter van het alfabet, de Z. Deze toetscombinatie wordt uitgesproken als *"kontrool zet"* en in teksten meestal aangeduid met ^Z of **CTL-Z**. Kortom: **^Z** betekent voor **consult** (en ook voor het later te bespreken **reconsult**) "einde van de file" (**end_of_file**). Na **^Z** verschijnt weer de normale Prologprompt.
Een voorbeeld: wanneer (1") en (2") aan de gegevensbank toegevoegd moeten worden vanaf het toetsenbord *kan* dat als volgt:

```
| ?-consult(user).
| mens(socrates).
| sterfelijk(X) :- mens(X).
| ^Z
| ?-
```

De methode om via het toetsenbord clausules in de gegevensbank te zetten is alleen

verantwoord om even tussendoor hele kleine programmaatjes uit te proberen, omdat er na het beëindigen van Prolog niets meer van overblijft. De beste methode voor het ontwikkelen van een programma is een programmafile te maken en deze te **(re)consult**eren.

Niets belet ons om de clausules van een file meer dan een keer in de gegevensbank te zetten. Meestal is dit niet de bedoeling en gebeurt dit per ongeluk. Het gevolg is dat we oplossingen dubbel krijgen, zoals we in de volgende sessie zien:

```
A> prolog
| ?-sterfelijk(M).
no
| ?-consult('socrates.pro').
yes
| ?-sterfelijk(M).
M = socrates;
no
| ?-consult('socrates.pro').
yes
| ?-sterfelijk(M).
M = socrates;
M = socrates
yes
| ?-halt.
A>
```

Het dubbel binnenhalen van clausules kan voorkomen worden door altijd **reconsult** te gebruiken.

2.8.2 reconsult

Evenals **consult** zet **reconsult** de in de file aangetroffen clausules in de gegevensbank. Echter, **reconsult** *herdefinieert* predikaten. Onder de *definitie van een predikaat* p *met talligheid* n verstaan we de verzameling clausules bestaande uit de *n*-plaatsige feiten die *p* als predikaat hebben, plus alle regels met een dergelijke propositie als hoofd. Bevat een file een definitie voor een n-plaatsig predikaat, dan zal de gegevensbank na een **reconsult** van die file alleen *die* definitie bevatten. Een eventueel al aanwezige definitie met dezelfde talligheid wordt opgeruimd. Het aantal clausules van de nieuwe definitie kan groter of kleiner zijn dan dat van de oude. Ook **reconsult** vereist een *atoom* als argument voor de filenaam

Een voorbeeld: Staan in de file *oud.pro* de clausules:

```
mens(jan).
mens(piet).
sterfelijk(X) :- dier(X).
woning(diogenes,ton).
```

En bevat de file *nieuw.pro* de clausules:

```
woning(X) :- huis(X).
sterfelijk(X) :- mens(X).
mens(diogenes).
```

dan is de volgende sessie mogelijk (we gebruiken **listing** om te volgen wat er met de gegevensbank gebeurt:

```
| ?-consult('oud.pro').
| ?-reconsult('nieuw.pro').
| ?-listing.
mens(diogenes).                 /* Enige mens! Vervangt alle oude clausules. */
sterfelijk(X) :- mens(X).       /* Vervangt oude clausule     */
woning(X) :- huis(X).           /* Toegevoegd (1 argument)    */
woning(diogenes,ton).           /* Ongewijzigd                */
yes
```

(Wat tussen /* en */ staat is slechts ter toelichting van de resultaten.)
De gegevensbank met dubbel gedefinieerde predikaten van het laatste voorbeeld van 2.8.1 kunnen we herstellen door de file *socrates.pro* te reconsulten.

2.8.3 Een alternatieve notatie

Voor **consult** en **reconsult** bestaat ook een *lijstnotatie*: bieden we als vraag aan een aantal filenamen tussen spekhaken ([en]), van elkaar gescheiden door komma's, dan wordt elk van de files ge**consult**eerd. Staat voor een filenaam een min-teken, dan wordt de file ge**reconsult**eerd. Ons voorbeeld met de files *oud.pro* en *nieuw.pro* had er ook als volgt uit kunnen zien:

```
| ?-['oud.pro'].
yes
| ?-[-'nieuw.pro'].
yes
```

of nog korter:

```
| ?-['oud.pro', -'nieuw.pro']
yes
```

Helaas valt het vaak niet mee de apostrophen niet te vergeten.

2.8.4 Directieven en pogingen systeempredikaten te herdefiniëren

Vaak komt het voor dat we *tijdens* het (re)**consult**eren van een file bepaalde doelproposities direct op afleidbaarheid worden onderzocht. Dit is mogelijk door de doelpropositie (of meer algemeen de *vraag*, zie hoofdstuk 3) in de file op te nemen, *voorafgegaan door een indien-symbool*. Een voorbeeld zien we hieronder. We nemen aan dat we een groot Prologprogramma hebben, dat verspreid is over drie files (*hoofd.pro, deel1.pro, deel2.pro*). Om dit programma te kunnen gebruiken moeten we dus alle drie de files re(**consult**)en. We hoeven echter slechts één keer een re(**consult**) opdracht te geven, wanneer we in de file *hoofd.pro* twee directieven opnemen die er voor zorgen dat de overige files binnengehaald worden:

```
:-reconsult('deel1.pro').
:-reconsult('deel2.pro').

/* Hierna volgt de rest van hoofd.pro */
```

Met één *directief* kan het natuurlijk ook:

```
:-[-'deel1.pro',-'deel2.pro'].
```

Systeempredikaten kunnen niet geherdefinieerd worden. Wanneer dat toch geprobeerd wordt krijgen we een foutmelding van het systeem. Wanneer we bijvoorbeeld in de bovengenoemde file *hoofd.pro* de twee *indien-symbolen vergeten*, dan bevat de file geen *directieven*, maar worden de twee aanwezige **reconsult** proposities opgevat als *een nieuwe definitie van* **reconsult**, hetgeen een foutmelding oplevert.

2.8.5 asserta en aanverwanten (inleiding)

Wanneer het om de toevoeging van slechts enkele clausules gaat, kan gebruik gemaakt worden van de predikaten **asserta**, **assertz** en **assert**. Deze predikaten worden echter alleen gebruikt voor speciale doeleinden en we zullen ze, om misbruik en een te imperatieve stijl van programmeren te voorkomen, pas veel later bespreken.

2.9 Commentaar en spaties

Vaak is het handig om programma's van commentaar te voorzien. Daarvoor bestaan de symbolen /* en */. Alles wat tussen deze twee symbolen staat wordt door de interpretator genegeerd, hoe ver ze ook uit elkaar staan. Commentaar wordt dus niet in de gegevensbank gezet. Commentaar mag overal staan waar ook spaties mogen staan. Spaties mogen staan *tussen* een propositie en een speciaal teken (haakje, :-, komma en puntkomma), tussen speciale tekens en ook tussen de diverse onderdelen van een propositie, maar *niet* tussen het predikaat en het haakje-openen van de argumentenlijst. Het volgende voorbeeld is dus *niet* goed:

```
mens   (socrates).
```

Overal waar spaties zijn toegestaan, mogen ook *returns* staan. Kortom: commentaar, spaties en returns worden gelijk behandeld (in het Engels worden ze tezamen *white space* genoemd). Een voorbeeld met (te) veel *white space* op plaatsen waar dat toegestaan is, staat hieronder:

```
/*
        Veel te veel wit, maar wel te consulteren
*/
mens(  socrates  ) .              /* socrates is een mens */
sterfelijk(  X  )  :-
        mens(  X  ).
```

Voor de leesbaarheid van een programma is het verstandig om geen commentaar en *returns* binnen een propositie te zetten. Er zijn meer regels die tot een goede lay-out van programma's leiden. We zullen ons best doen de voorbeelden in dit boek een voorbeeldige lay-out te geven.

Blijken delen van een programma na een **consult** niet in de gegevensbank te staan, ondanks het feit dat ze wel in de programmafile staan, kijk dan de commentaartekens goed na!

Bijvoorbeeld: een **consult** van de volgende file voegt niets toe aan de gegevensbank, aan - gezien alle clausules (al of niet met opzet) tot commentaar gereduceerd zijn.

```
/*    mens(jan).
      mens(piet).
      sterfelijk(X) :- dier(X).
      woning(diogenes,ton).  */
```

Alles wat hiervoor over *white space* in clausules gezegd werd, geldt mutatis mutandis ook voor *vragen*, zoals al bleek aan het einde van 2.4.

2.10 Opgaven

2.1 Zie sectie 1.5 van het vorige hoofdstuk. Zet met behulp van een editor de clausules (1"), (4") en (2") in de file *socrates.pro*. Het MS-DOS commando *type* maakt het mogelijk om de inhoud van een file op het scherm te bekijken. Dus:

```
A> type socrates.pro
```

moet na het editen te zien geven:

```
mens(socrates).
mens(russell).
sterfelijk(X) :- mens(X).
```

Start Prolog en **reconsult** *socrates.pro* Bied vervolgens alle doelproposities aan die in sectie 1.5 besproken werden en vergelijk de resultaten met die in 1.5. Doorloop de programma - ontwikkelcyclus die in sectie 2.1 beschreven werd, net zo lang tot de goede antwoorden verkregen zijn.

2.2 Maak een file *huisdier.pro,* die de clausules van de gegevensbank van opgaven 1.1 en 1.2 bevat. Beantwoord vervolgens de vragen van deze opgaven met behulp van de computer.

2.3 Maak de files *oud.pro* en *nieuw.pro* zoals beschreven in 2.8.2, en voer de daar beschreven sessie uit. Probeer ook de lijstnotatie van 2.8.3.

2.4 Gebruik de mogelijkheid een programma te becommentariëren om de clausules in de file *socrates.pro* van de in de tekst gebruikte nummers, (1", etc.), te voorzien. Voeg aan het begin van de file een of meer regels toe, waarin er melding van wordt gemaakt dat het om het eerste Prologprogrammaatje gaat. Probeer daarna of alles nog net zo werkt als in opgave 2.1.

2.5 (a) Voeg aan de file *huisdier.pro* een clausule toe die er voor zorgt dat honden tot de dieren gerekend worden (dus: X is een ***dier*** indien X een ***hond*** is). Doe hetzelfde voor de poezen. Deze uit twee regels bestaande definitie van het predikaat ***dier*** behelst dus dat X een ***dier*** is, wanneer X een ***hond*** of een ***poes*** is.
(b) Voeg een derde clausule toe, die dieren ***sterfelijk*** maakt.
(c) Onderzoek met behulp van de interpretator de gevolgen van de wijzigingen.
(d) Bekijk eens wat het effect is op de volgorde waarin de oplossingen voor de doelpropositie **sterfelijk(X)** geproduceerd worden, wanneer de twee clausules die het predikaat ***dier*** definiëren van plaats verwisseld worden.
(e) Als we naast de huisdieren ook mensen als sterfelijke wezens in onze gegevensbank willen hebben, moeten we dan, nadat *huisdier.pro* in de gegevenbank gezet is, de file *socrates.pro* binnenhalen met **consult** of met **reconsult**?

3 Het inferentiemechanisme, deel II

In alle voorbeelden en opgaven tot nu toe hebben we de Prologinterpretator alleen maar vragen gesteld die uit één propositie bestonden. Ook tijdens het afleidingsproces leidde elke doelpropositie na toepassing van een regel hoogstens tot onderzoek naar de afleidbaarheid van één andere propositie. Het afleidingsmechanisme is echter veel krachtiger: het is mogelijk proposities met elkaar te combineren tot *logische formules* met behulp van de operatoren voor *conjunctie* (*logisch **en***) en *disjunctie* (*logisch **of***). Conjunctie wordt aangegeven door een *komma*; disjunctie door een *puntkomma*. *Haakjes* kunnen worden gebruikt om de interpretatie te beïnvloeden.

De vormingsregels voor logische formules zijn:
(1) als **P** een propositie is, dan is **P** een logische formule;
(2) als **F1** en **F2** logische formules zijn, dan is **F1,F2** een logische formule (een *conjunctie*);
(3) als **F1** en **F2** logische formules zijn, dan is **F1;F2** een logische formule (een *disjunctie*);
(4) als **F** een logische formule is, dan is ook **(F)** een logische formule.

Voorbeelden van samengestelde logische formules en hun mogelijke interpretaties zijn:

hond(X),poes(X)	X is een hond *en* X is een poes
hond(X),poes(Y)	X is een hond *en* Y is een poes
hond(X);poes(X)	X is een hond *of* X is een poes
hond(X);poes(Y)	X is een hond *of* Y is een poes
vader(G,O),ouder(O,K)	G is de vader van O *en* O is een ouder van K (dus G is een grootvader van K)
vader(G,O),(moeder(O,K);vader(O,K))	G is de vader van O *en* O is de moeder *of* de vader van K (G is dus weer grootvader van K)

Samengestelde logische formules kunnen voorkomen als *vraag* en als *rechterkant van een regel*. (Een feit en het hoofd van een regel bestaan altijd uit de simpelste logische formule, namelijk een propositie.) Voor *variabelen in clausules* blijft gelden dat het *bereik* gelijk is aan de clausule waarin ze voorkomen. Voor *variabelen in een vraag* is het *bereik* gelijk aan die vraag. Alleen een unificatiestap kan variabelen met een verschillend bereik met elkaar in verband brengen.

In wat volgt zullen we nader ingaan op de wijze waarop de Prologinterpretator met logische formules omspringt.

3.1 Conjunctie

3.1.1 Conjunctie in vragen

Een conjunctie is afleidbaar indien elk van haar leden (*conjuncten*) afleidbaar is. De interpretator werkt van links naar rechts. Zodra een conjunct niet afleidbaar blijkt, wordt de afleiding beëindigd en rapporteert de interpretator *no*. Blijken alle conjuncten afleidbaar, dan is de hele conjunctie afleidbaar, hetgeen het antwoord *yes* oplevert, eventueel nadat een of meer mogelijke oplossingen voor variabelen op het scherm getoond zijn. Uit de vormingsregels blijkt dat een conjunct de vorm van een propositie, een conjunctie of een disjunctie kan hebben.
(1) Bestaat een conjunct uit een propositie, dan is het conjunct afleidbaar als de propositie afleidbaar is.
(2) Bestaat een conjunct zelf uit een conjunctie, dan is het conjunct afleidbaar als elk van haar conjuncten afleidbaar is.
(3) Bestaat een conjunct uit een disjunctie, dan is het conjunct afleidbaar als minstens één van de disjuncten afleidbaar is (zie 3.2).
De meest voorkomende logische formule in Prologprogramma's is de conjunctie waarbij elk van de conjuncten een propositie is. Geval (2) doet zich alleen voor bij overbodig gebruik van haakjes. Geval (3) moet ten behoeve van de begrijpelijkheid van programma's met mate toegepast worden (zie 3.2 en 6.1).

3.1.2 Backtracking

Om het onderzoek naar de afleidbaarheid van een conjunctie te beschrijven, moeten we dieper ingaan op het inferentiemechanisme, met name op het proces van *backtracking*.

Bij het zoeken naar een oplossing van een vraag maakt Prolog gebruik van *depth-first search* in combinatie met *backtracking*. Wat dat inhoudt, zullen we aan de hand van een voorbeeld duidelijk maken. We gaan uit van een woordenboekje in de gegevensbank dat uit de volgende clausules bestaat (toevallig allemaal feiten):

```
persoonlijk_voornaamwoord(ik,1,enkelvoud).
persoonlijk_voornaamwoord(zij,3,enkelvoud).
persoonlijk_voornaamwoord(zij,3,meervoud).

koppelwerkwoord(ben,1,enkelvoud).
koppelwerkwoord(is,3,enkelvoud).
koppelwerkwoord(zijn,3,meervoud).

zelfstandig_naamwoord(programmeur,enkelvoud).
zelfstandig_naamwoord(psycholoog,enkelvoud).
zelfstandig_naamwoord(buren,meervoud).
```

We bespreken het gedrag van de interpretator voor de volgende conjunctie, waarin we vragen twee woorden (*W* en *Z*) te zoeken, die tezamen met het al gespecificeerde persoonlijk voornaamwoord *zij* een correct zinnetje vormen:

```
| ?- persoonlijk_voornaamwoord(zij,Persoon,Getal),
     koppelwerkwoord(W,Persoon,Getal),
     zelfstandig_naamwoord(Z,Getal).
```

Voor de correctheid van de zin moeten de drie woorden qua ***Persoon*** en ***Getal***

overeenstemmen. De eerste (maar niet de enige) oplossing voor de vier variabelen in de vraag is:

Persoon=3
Getal=enkelvoud
W=is
Z=programmeur

Het eerste zinnetje dat aan de eisen voldoet is dus: *zij is programmeur.*

Wat er na het stellen van de vraag gebeurt, is het volgende.

De eerste propositie van de conjunctie heeft tot gevolg dat er gezocht gaat worden naar een propositie met als predikaat[1] **persoonlijk_voornaamwoord/3**, dat als eerste argument het woord ***zij*** bevat. Er bevindt zich inderdaad zo'n predikaat in de gegevensbank, waarbij het tweede en derde argument respectievelijk ***3*** en ***enkelvoud*** zijn. De doelpropositie had als tweede en derde argument de variabelen ***Persoon*** en ***Getal***, hetgeen leidt tot de substitutie:

Persoon=3, Getal=enkelvoud.

Het bereik van een variabele in een vraag is gelijk aan die vraag, dus de rest van de conjunctie luidt na de unificatiestap:

koppelwerkwoord(W,3,enkelvoud),
zelfstandig_naamwoord(Z,enkelvoud).

Op dit punt had de interpretator alle volgende oplossingen voor het eerste conjunct kunnen zoeken. Maar Prologinterpretatoren werken altijd *depth-first,* wat wil zeggen dat na het vinden van een deelresultaat niet eerst gezocht wordt naar alternatieven om tot datzelfde deelresultaat te komen (*breadth-first*), maar dat geprobeerd wordt nog een stap dichter bij de totaaloplossing te komen. De interpretator zal, nadat het eerste conjunct tot een oplossing geleid heeft, een *stap vooruit* doen naar het volgende conjunct. Er zal dus een afleiding worden gezocht voor de tweede propositie, die het koppelwerkwoord van de zin moet opleveren. Het zal duidelijk zijn dat dat leidt tot de enige oplossing voor ***W***:

W=is

Vervolgens gaat de interpretator weer een *stap vooruit* en komt de derde propositie aan de beurt. Dat leidt tot de eerst mogelijke oplossing voor het zelfstandig naamwoord **Z**:

Z=programmeur.

Nu alle conjuncten afleidbaar zijn gebleken, verschijnt de oplossing op het scherm:

Persoon=3
Getal=enkelvoud
W=is
Z=programmeur

[1] *Voortaan betekent de toevoeging /**n** bij een predikaat, dat dat predikaat n argumenten heeft.*

Wat er verder gaat gebeuren hangt af van hetgeen we intypen. Typen we een *return* in, dan is daarmee de vraag afgehandeld en verschijnt *yes* op het scherm.

Geven we door een puntkomma aan dat we meer oplossingen willen, dan wordt de laatste unificatiestap ongedaan gemaakt (we zullen dit een *deünificatiestap* noemen) en er wordt gezocht naar een alternatieve oplossing voor de laatste doelpropositie.

In ons voorbeeld betekent dit, dat voor de laatst gebonden variabele (Z) een volgende oplossing gezocht wordt. Let wel: de overige variabelen (***Persoon*** en ***Getal***) blijven hun waarden houden, want die bindingen waren afkomstig van een substitutie behorende bij een andere (namelijk de eerste) propositie. Er is een andere oplossing voor **Z,** namelijk:

Z=psycholoog.

Z wordt opnieuw gebonden en de interpretator rapporteert de tweede oplossing:

Persoon=3
Getal=enkelvoud
W=is
Z=psycholoog

Typen we nu weer een puntkomma in, dan wordt na een deünificatiestap nog een oplossing voor **Z** gezocht. Deze wordt niet gevonden, omdat de mogelijkheden voor het derde conjunct helemaal uitgeput zijn.
Daarom gaat de interpretator over tot een *backtrackingstap*: de interpretator doet in de conjunctie *een stap terug*. Dit houdt in dat naar nieuwe afleidingen voor het conjunct voorafgaand aan het huidige, gezocht wordt. In ons voorbeeld wordt dus naar een alternatieve afleiding voor de koppelwerkwoordspropositie gezocht. Hiertoe wordt de op dit moment geldende substitutie ***W=is*** ongedaan gemaakt (niet die voor ***Persoon*** en ***Getal***!) en er wordt een nieuwe waarde voor ***W*** gezocht. Er staat echter maar één koppelwerkwoord derde persoon enkelvoud in de gegevensbank, dus er is geen alternatieve afleiding voor het tweede conjunct.

Daarom volgt er weer een *backtrackingstap*, waardoor de interpretator belandt bij de eerste propositie. Bij deze propositie hoort de substitutie voor de variabelen ***Persoon*** en ***Getal***. Deze wordt ongedaan gemaakt en er wordt een alternatieve oplossing voor het eerste conjunct gezocht en gevonden, namelijk:

Persoon=3, Getal=meervoud

Na deze alternatieve oplossing voor het eerste conjunct gevonden te hebben doet de interpretator weer een *stap vooruit* , waardoor de tweede propositie weer aan de beurt komt, waarbij uiteraard rekening gehouden wordt met de nieuwe bindingen. Dit geeft de volgende binding:

W=zijn

Vervolgens komt de derde propositie weer aan de beurt, wat leidt tot een nieuwe oplossing voor de gehele conjunctie:

Persoon=3
Getal=meervoud
W=zijn
Z=buren

Hierna wacht de interpretator weer.

Een volgende puntkomma leidt de interpretator via twee backtrackingstappen terug naar de eerste propositie, welke ook geen alternatieve oplossing meer levert, zodat onze vraag afgehandeld is.

3.1.3 Conjunctie in clausules

Zoals gezegd kunnen samengestelde logische formules ook voorkomen als conditie in een regel. We kunnen bijvoorbeeld de complexe vraag uit ons laatste voorbeeld gebruiken in een clausule die het predikaat **zij_zinnetje/2** definieert:

```
zij_zinnetje(W,Z):-
        persoonlijk_voornaamwoord(zij,Persoon,Getal),
        koppelwerkwoord(W,Persoon,Getal),
        zelfstandig_naamwoord(Z,Getal).
```

Als bovenstaande clausule zich in de gegevensbank bevindt, dan leidt de vraag

```
| ?-zij_zinnetje(W2,W3).
```

tot vrijwel hetzelfde zoekproces als hierboven beschreven werd voor de conjunctie. Alleen aan het begin en aan het eind komt er een stap bij.

Eerst wordt gezocht naar een clausule in de gegevensbank die unificeerbaar is met de propositie uit de vraag. Deze wordt gevonden in de vorm van de eerste (en enige) clausule van de definitie van **zij_zinnetje**. Dit leidt tot de substitutie:

W2=W, W3=Z

Hierna verloopt alles hetzelfde als hierboven beschreven. Wanneer voor de conjunctie een oplossing gevonden is, is aan de condities voor afleidbaarheid van het hoofd van de clausule voldaan en daarmee is voor de doelpropositie uit de vraag een afleiding gevonden. De interpretator rapporteert de oplossing voor *de variabelen uit de vraag* op het scherm:

W2=is
W3=programmeur

(***Persoon*** en ***Getal*** fungeren in deze afleiding alleen "intern".) Na een puntkomma volgt weer een deünificatiestap voor de laatste propositie in de afleiding, gevolgd door het vinden van de tweede en de derde oplossing, zoals eerder beschreven is.

Nadat alle mogelijkheden voor de conjunctie uitgeput zijn, komt de interpretator weer bij het hoofd van de eerste clausule van de definitie van het predikaat **zij_zinnetje**/2 uit. De substitutie

W2=W, W3=Z

wordt ongedaan gemaakt en *de interpretator zoekt naar de volgende clausule van de definitie van het predikaat.* In het voorbeeld ontbreekt deze. Was er nog een met de doelpropositie uit de vraag unificeerbare clausule geweest, dan was het zoeken naar oplossingen doorgegaan.

Niets verbiedt ons een propositie met een predikaat dat door middel van een conjunctie gedefinieerd is, zelf ook weer in een logische formule te gebruiken. Zo'n propositie fungeert dan eenvoudigweg als een conjunct of disjunct waarvan de afleidbaarheid moet worden onderzocht. En dit geschiedt niet anders dan zojuist beschreven is. Het in elkaar uitdrukken van predikaten met behulp van regels is uit het oogpunt van programmaontwikkeling erg belangrijk, omdat dit het mogelijk maakt een groot probleem te *reduceren* tot een aantal deelproblemen, die alle ook weer gereduceerd worden, enzovoorts, net zo lang tot we aangeland zijn bij de problemen die Prolog direct op kan lossen. Deze methode van programmeren heet *Top-Down programmeren.* Om deze methode toe te kunnen passen moeten we echter wel weten welke problemen in Prolog direct oplosbaar zijn en daarvoor dienen boeken als het onderhavige.

3.1.4 Volgen van een afleiding met de *tracer*

Elke Prologimplementatie biedt meer of minder geavanceerde hulpmiddelen voor het opsporen van fouten in programma's (*debugging tools*). Een hulpmiddel dat nooit ontbreekt is een *tracer*. *Tracen* ("tresen") wil zeggen dat alle stappen van een afleiding, of een selectie daaruit, op het scherm gevolgd kunnen worden (*trace*=spoor). Hoe een *tracer* aan- en uitgezet wordt, wat voor informatie over een stap op het scherm verschijnt, hoe delen van een afleiding (*call*, *redo*, *exit* en/of *fail*) onzichtbaar gelaten kunnen worden met behulp van het **leash** predikaat, hoe alleen die predikaten gevolgd kunnen worden waarin een fout vermoed wordt (**spy** en **nospy**), verschilt per Prologimplementatie. Voor dit alles zij men naar de handleiding van de gebruikte interpretator verwezen. In hoofdlijnen produceren alle tracers dezelfde informatie. Alle *tracers* wachten na elke stap van de getoonde afleiding op een eenletter commando dat aangeeft wat er verder moet gebeuren. Er kan dan minstens gekozen worden uit:

- doe de volgende stap (meestal aangegeven met een *return*)
- breek de afleiding af (meestal *abort*)
- neem *fail* als resultaat van de afgebeelde doelpropositie aan (*fail*)
- stop de tracer (niet de afleiding) (*notrace*)
- toon de afleiding voor de getoonde doelpropositie niet (*skip*)
- geef een overzicht van de mogelijke commando's (*help* of *?*)

In dit boek zullen we van al deze fraaiïgheden geen gebruik maken en houden we het eenvoudig. We zullen in voorbeeldsessies soms twee (niet in elke Prolog implementatie gelijk gedefinieerde) systeempredikaten gebruiken:

trace	zet de *tracer* aan
notrace	zet de *tracer* uit.

Hieronder geven we een *trace* van het voorbeeld dat in 3.2 besproken werd. We beginnen met het aanzetten van de tracer. De dubbele streep in de daarop volgende prompt dient om ons eraan te herinneren dat we in de *debugging-mode* zitten. Daarna stellen we de vraag waarvan we de afleiding willen bestuderen:

```
| ?-trace.
Debug mode on.
yes

|| ?-zij_zinnetje(W2,W3).
[1] 0 Call: zij_zinnetje(W2,W3) ?
Match: zij_zinnetje(W2,W3):-
        persoonlijk_voornaamwoord(zij,Persoon,Getal),
        koppelwerkwoord(W2,Persoon,Getal),
        zelfstandig_naamwoord(W3,Getal).
        [2] 1 Call: persoonlijk_voornaamwoord(zij,Persoon,Getal) ?
        Match: persoonlijk_voornaamwoord(zij,3,enkelvoud).
        [2] 1 Exit: persoonlijk_voornaamwoord(zij,3,enkelvoud)
        [3] 1 Call: koppelwerkwoord(W2,3,enkelvoud) ?
        Match: koppelwerkwoord(is,3,enkelvoud).
        [3] 1 Exit: koppelwerkwoord(is,3,enkelvoud)
        [4] 1 Call: zelfstandig_naamwoord(W3,enkelvoud) ?
        Match: zelfstandig_naamwoord(programmeur,enkelvoud).
        [4] 1 Exit: zelfstandig_naamwoord(programmeur,enkelvoud)
[1] 0 Exit: zij_zinnetje(is,programmeur)
W2=is,
W3=programmeur;
        [4] 1 Pop: zelfstandig_naamwoord(programmeur,enkelvoud)
        [4] 1 Redo: zelfstandig_naamwoord(W3,enkelvoud) ?
        Match: zelfstandig_naamwoord(psycholoog,enkelvoud).
        [4] 1 Exit: zelfstandig_naamwoord(psycholoog,enkelvoud)
[1] 0 Exit: zij_zinnetje(is,psycholoog)
W2=is,
W3=psycholoog;
        [4] 1 Pop: zelfstandig_naamwoord(psycholoog,enkelvoud)
        [4] 1 Redo: zelfstandig_naamwoord(W3,enkelvoud) ?
        [4] 1 Fail: zelfstandig_naamwoord(W3,enkelvoud)
        [3] 1 Pop: koppelwerkwoord(is,3,enkelvoud)
        [3] 1 Redo: koppelwerkwoord(W2,3,enkelvoud) ?
        [3] 1 Fail: koppelwerkwoord(W2,3,enkelvoud)
        [2] 1 Pop: persoonlijk_voornaamwoord(zij,3,enkelvoud)
        [2] 1 Redo: persoonlijk_voornaamwoord(zij,Persoon,Getal) ?
        Match: persoonlijk_voornaamwoord(zij,3,meervoud).
        [2] 1 Exit: persoonlijk_voornaamwoord(zij,3,meervoud)
        [3] 1 Call: koppelwerkwoord(W2,3,meervoud) ?
        Match: koppelwerkwoord(zijn,3,meervoud).
        [3] 1 Exit: koppelwerkwoord(zijn,3,meervoud)
        [4] 1 Call: zelfstandig_naamwoord(W3,meervoud) ?
        Match: zelfstandig_naamwoord(buren,meervoud).
        [4] 1 Exit: zelfstandig_naamwoord(buren,meervoud)
[1] 0 Exit: zij_zinnetje(zijn,buren)
W2=zijn,
W3=buren
yes

|| ?-notrace.
Debug mode off.
yes
```

Verklaring van de door de tracer gebruikte termen

Call (Try) toont de nieuwe doelpropositie die op afleidbaarheid onderzocht moet worden. De nieuwe propositie krijgt een ***volgnummer** tussen rechte haken toegewezen:* **[*getal*]**. Dit volg - nummer gebruikt de tracer elke keer wanneer er iets over deze doelpropositie gemeld wordt. De interpretator begint een afleiding te zoeken voor de afgebeelde doelpropositie indien na het afgebeelde *vraagteken* een *return* wordt ingetypt, wat in het voorbeeld in alle gevallen gedaan werd. Na het volgnummer staat een tweede ***getal*** dat aangeeft via hoeveel doelproposities (*ancestors*, ***voorouders***) de nu aan de beurt zijnde doelpropositie geactiveerd werd. Dit getal is dus gelijk aan het aantal regels dat toegepast werd voordat deze propositie bereikt werd of anders gezegd, gelijk aan het aantal *reductiestappen* dat tot deze doelpropositie geleid heeft.

Match De interpretator heeft een clausule gevonden waarvan het hoofd *unificeerbaar* is met de doelpropositie.

Exit (Succ) Er is een oplossing gevonden: de gevonden waarde(n) zijn gezet op de plaats waar variabelen stonden. Met andere woorden, er is een *stap vooruit* gezet en er is een *substitutie* gevonden.

Pop Geeft aan dat een *deünificatie* zal plaatsvinden.

Redo (Retry) Er wordt een *volgende oplossing* voor de afgebeelde doelpropositie gezocht.

Fail Er is *geen afleiding (meer)* voor de getoonde doelpropositie.

3.1.5 Opgaven

3.1 Zet het woordenboekje uit sectie 3.1 en de definitie voor **zij_zinnetje** in de file *zinnetje.pro*. Probeer de in dit hoofdstuk beschreven voorbeelden uit op de computer. Maak voor het volgen van afleidingen gebruik van de *tracer*. Breng vervolgens de volgende wijzigingen aan in de file.

(a) Completeer het lijstje van persoonlijke voornaamwoorden in de file. Voeg ook de ont - brekende vormen van het koppelwerkwoord *zijn* toe aan het woordenboekje.

(b) Definieer het predikaat **zinnetje/3** dat met behulp van het woordenboekje zijn argumenten unificeert met woorden die tezamen een zinnetje vormen. Voorbeelden van gebruik zien we in de volgende sessie:

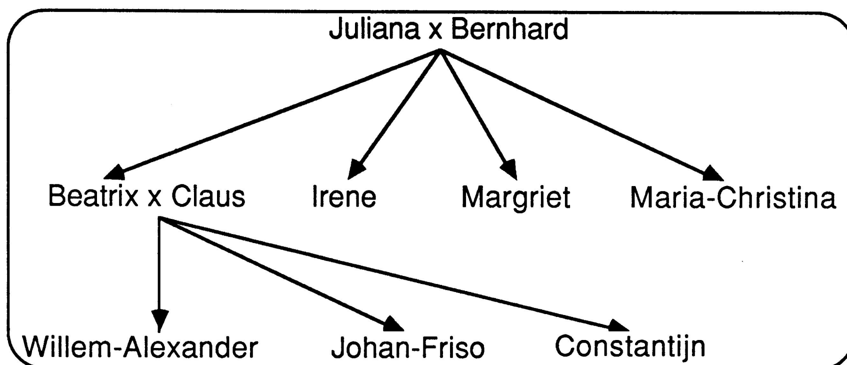
```
| ?-zinnetje(P,W,Z).
P=ik
W=ben
Z=programmeur;
P=ik
W=ben
Z=psycholoog
yes
| ?-zinnetje(P,bent,psycholoog).
P=jij;
no
| ?-zinnetje(jullie,bent,buren).
no
```

(c) Maak, in de file *zinnetje.pro,* een kopie van het predikaat **zinnetje.** Wijzig in de kopie het predikaat in het hoofd van de regel in **vraagje.** Breng vervolgens in **vraagje** de nodige veranderingen aan, opdat het predikaat zijn naam eer aan doet. Hierbij hoeft de behandeling van vraagzinnetjes met *jij* (voorlopig) niet foutloos te zijn, dus wordt bijvoorbeeld *Bent jij programmeur?* goedgekeurd.

(d) Definieer een predikaat **zinnetje_vraagje**/6 dat checkt of de eerste drie argumenten een mededelende driewoordzin vormen en dat daarna de woorden van die zin in de volgorde van de vraagzin unificeert met de drie andere argumenten. Voorbeelden van het gebruik van het nieuwe predikaat zijn de volgende:

| ?-**zinnetje_vraagje(zij,zijn,buren,A,B,C).**
A=zijn
B=zij
C=buren;
no
| ?-**zinnetje_vraagje(zij,A,buren,zijn,B,C).**
A=zijn
B=zij
C=buren
yes
| ?-**zinnetje_vraagje(A,B,C,B,A,C).**
A=ik
B=ben
C=programmeur
yes
| ?-**zinnetje_vraagje(A,B,C,A,B,C).**
no
| ?-**zinnetje_vraagje(jij,B,C,D,E,F).**
B=bent
C=programmeur
D=bent
E=jij
F=programmeur
yes

3.2 Representeer de volgende stamboom met tien personen in de file *stamboom.pro.*



Voor het weergeven van de ouder-kind relaties is het voldoende *feiten* te definiëren die alle van één van de volgende drie *typen* zijn (huwelijksrelaties behoeven niet weergegeven te worden):

is_man(*naam*)	*naam* is een man (5)
is_vrouw(*naam*)	*naam* is een vrouw (5)
kind(*naam1*,*naam2*)	*naam1* is een kind van *naam2 (14)*

Definieer nu *regels* voor de volgende familierelaties. Gebruik de 24 feiten van de stamboom om de definities uit te proberen.

zoon(Z,O)	Z is de zoon van O
dochter(D,O)	D is de dochter van O
ouder(O,K)	O is een ouder van K
vader(V,K)	V is de vader van K
moeder(M,K)	M is de moeder van K
ouders(V,M,K)	V en M zijn de ouders van K
grootouder(G,K)	G is grootouder van K
grootvader(G,K)	G is grootvader van K
grootmoeder(G,K)	G is grootmoeder van K
kleinkind(K,G)	K is kleinkind van G
kleinzoon(K,G)	K is kleinzoon van G
kleindochter(K,G)	K is kleindochter van G

Uiteraard mogen de predikaten naar elkaar verwijzen. Hieronder volgt een klein deel van de *oplossing*:

```
is_man(claus).
is_man(willem_alexander).

kind(willem_alexander,claus).

vader(V,K):-kind(K,V),is_man(V).
grootvader(G,K):-vader(G,X),ouder(X,K).
```

Voorbeeldvragen staan hieronder.

```
| ?-is_man(willem_alexander).
yes
| ?-vader(V,willem_alexander).
V=claus
yes
| ?-vader(willem_alexander,K).
no
| ?-grootvader(G,willem_alexander).
G=bernhard;
no
```

Om clausules voor de familierelaties *zus* en *broer* correct te definiëren, is nog wat meer kennis van Prolog vereist. Probeer eens te achterhalen wat het probleem is.

3.2 Disjunctie

Disjunctie wordt weergegeven door een puntkomma. Evenals conjunctie kan disjunctie worden toegepast in vragen en in de condities van een clausule. Een disjunctie is afleidbaar als één van de disjuncten afleidbaar is. Bij het onderzoeken van de afleidbaarheid werkt de interpretator de disjuncten één voor één van links naar rechts af. Zodra een disjunct afleidbaar blijkt, is de hele disjunctie afleidbaar en wordt er voor de in de doelpropositie aanwezige variabelen een oplossing op het scherm getoond. Maar Prolog zou zijn non-deterministische karakter verliezen, indien na het intypen van een puntkomma niet naar een volgende oplossing zou worden gezocht. Eerst wordt gezocht naar alternatieve afleidingen voor het disjunct waarmee de interpretator bezig was. Pas nadat die allemaal gevonden zijn, wordt het volgende disjunct geprobeerd. Een disjunctie is altijd om te zetten in een aantal "losse" clausules.
Bijvoorbeeld:

```
vrucht(X):-
        appel(X);peer(X);kiwi(X).
```

betekent dat X een vrucht is, als X een appel, een peer of een kiwi is. Deze ene clausule is equivalent met de volgende clausules (in de gegeven volgorde!):

```
vrucht(X):-appel(X).
vrucht(X):-peer(X).
vrucht(X):-kiwi(X).
```

Een disjunctie kan onderdeel zijn van een conjunctie; in dat geval is invoering van een extra predikaat vaak op zijn plaats. In plaats van:

```
maaltijd(V,H,F,K):-
        voorgerecht(V),
        hoofdgerecht(H),
        (appel(F);peer(F);kiwi(F)),
        koffie(K).
```

is overzichtelijker (zeker als het aantal **vrucht**en groter is):

```
maaltijd(V,H,F,K):-
        voorgerecht(V),
        hoofdgerecht(H),
        vrucht(F),
        koffie(K).
```

De definitie van de overige gangen wordt aan de fantasie van de lezer overgelaten.

Een disjunctie kan een conjunctie als disjunct bevatten. Ook in dit geval worden de disjuncten van links naar rechts verwerkt; als de conjunctie aan de beurt is, wordt de procedure gevolgd zoals beschreven in 3.1.

3.3 Haakjes en volgorde van conjunctie en disjunctie

We zagen hiervoor dat conjuncties disjuncties, en disjuncties conjuncties kunnen bevatten. We hebben nog een regel nodig die bepaalt of een samengestelde formule uiteindelijk een conjunctie of een disjunctie is. Die regel luidt: *conjunctie gaat voor op disjunctie.* Hiermee wordt bedoeld, dat bij het combineren van formules tot grotere formules de "aantrekkingskracht" (*prioriteit*, zie hoofdstuk 10), van de conjunctieoperator op een formule groter is dan die van de disjunctieoperator. Het gevolg is, dat de proposities in een formule eerst zoveel mogelijk tot conjuncties worden gecombineerd en pas daarna tot een disjunctie. Hierbij wordt een formule tussen haakjes als een ondoordringbaar geheel beschouwd; voor de proposities tussen de haakjes geldt dezelfde volgorderegel. In hoofdstuk 10 wordt dit thema formeler besproken; hier volstaan we met wat voorbeelden, die niet bedoeld zijn om het gebruik van complexe formules aan te moedigen:

Formule zonder haakjes	*Equivalente formule met haakjes*
a,b,c,d;e	**(a,b,c,d);e**
a,b,c;d,e	**(a,b,c);(d,e)**
a,b,c,(d;e)	**(a,b,c),(d;e)**
a,b;c,d;e	**(a,b);(c,d);e**
a,(b;c),d,e	**a,(b;c),(d,e)**
(a,b,c,d;e);(a,(b;c),d,e)	**((a,b,c,d);e);(a,(b;c),(d,e))**

Een voorbeeld met meer inhoud zien we in het rechterdeel van onderstaande clausule, waarbij elk disjunct bestaat uit een conjunctie, waarvan het tweede conjunct ook weer een disjunctie is:

```
grootouder(X,Y):-
        vader(T,Y),(vader(X,T);moeder(X,T));
        moeder(T,Y),(vader(X,T);moeder(X,T)).
```

Voor een correcte interpretatie mag geen van de haakjes worden weggelaten! Overzichtelijker is de volgende disjunctie, waarbij elk disjunct uit een eenvoudige conjunctie bestaat.

```
grootouder(X,Y):-
        vader(T,Y),vader(X,T);
        vader(T,Y),moeder(X,T);
        moeder(T,Y),vader(X,T);
        moeder(T,Y),moeder(X,T).
```

Het kan ook in vier aparte regels.

```
grootouder(X,Y):-vader(T,Y),vader(X,T).
grootouder(X,Y):-vader(T,Y),moeder(X,T).
grootouder(X,Y):-moeder(T,Y),vader(X,T).
grootouder(X,Y):-moeder(T,Y),moeder(X,T).
```

Formules waarin veel haakjes nodig zijn om misinterpretatie te voorkomen, leiden tot programma's als kaartenhuizen. Verderop zullen we voorbeelden van programmeerproblemen zien waarbij gebruik van de disjunctieoperator wél verdedigbaar is.

3.3.1 Opgave

3.3 Onderzoek met de *tracer* hoe Prolog de drie laatste voorbeelden behandelt.

3.4 Over de WAM en LIPS.

De hierboven gegeven beschrijving van het inferentiemechanisme diende slechts om een gebruiker van Prolog een abstract model te bieden op basis waarvan programma's ontworpen kunnen worden. Het gegeven model is als beschrijving van de werking van een Prolog - interpretator hoogst onvoldoende. Er bestaat wel een tot in alle details uitgewerkte abstracte Prologmachine, de*Warren Abstract Machine*, op congressen meestal afgekort, en op zijn Engels uitgesproken, aangeduid als de *WAM* (Warren, 1983). David Warren heeft in de 70'er jaren meegewerkt aan de ontwikkeling van de eerste Prologinterpretator op een DEC-10 computer die niet onaanvaardbaar traag was. Uit de versie van Prolog die door die interpretator verwerkt werd is, mede onder invloed van Clocksin en Mellish (1981), de als standaard aanvaarde versie van Prolog ontstaan (Edinburgh Prolog). De *WAM* is een papieren machine waarvan de basisinstructies zodanig gekozen zijn, dat Prologprogramma's er gemakkelijk en efficiënt in uitgedrukt kunnen worden. Vele thans beschikbare Prologimplementaties zijn software-implementaties van de *WAM.* Elke implementator brengt daarbij zijn eigen optimalisaties aan om de snelheid te vergroten. Deze snelheid wordt meestal uitgedrukt in de pseudo-exacte maat *LIPS*, wat staat voor *Logical Inferences Per Second.* Een *logical inference* correspondeert met wat wij in hoofdstuk 1 een *reductiestap* genoemd hebben, d.w.z. een *logical inference* correspondeert met de toepassing van een regel. Maar meer factoren, zoals het aantal variabelen en de snelheid waarmee de unificatiestappen uitgevoerd worden, beïnvloeden de totale snelheid. Daarom moet altijd in detail gekeken worden naar de testprogramma's op grond waarvan het aantal *LIPS* berekend is, om een juiste interpretatie van de opgegeven snelheid te verkrijgen. Het probleem is vergelijkbaar met de bepaling van het benzineverbruik van auto's. Voor een introductie in de*WAM* zij men verwezen naar (Gabriel e.a., 1986).

4 Complexe argumenten, =, \= en _.

Een argument van een propositie kan behalve uit een *constante* of *variabele* ook bestaan uit een *anonieme variabele*, uit een *structuur*, of uit een *lijst*. *Constanten, (anonieme) variabelen, structuren* en *lijsten* worden gezamenlijk aangeduid met de term *term*. We kunnen dus kortweg formuleren: een argument van een propositie bestaat uit een term. Om beter de mogelijkheden van complexe argumenten te kunnen illustreren, introduceren we eerst de *unificatie-operator*.

4.1 De unificatieoperator(=) en zijn tegenhanger(\=)

Unificatie kan expliciet worden gebruikt in vragen en in definities van predikaten, namelijk door middel van de *unificatieoperator*. Een *operator* is een één- of tweeplaatsig predikaat dat syntactisch afwijkt van een normaal predikaat. (We komen in de hoofdstukken 6 en 10 terug op het begrip *operator*.) De *unificatieoperator* is tweeplaatsig en wordt voorgesteld door een *is-gelijk-teken*, =, dat *tussen* de twee argumenten geplaatst wordt:

```
| ?-Term1=Term2.
```

De prologinterpretator zal bij een doelpropositie zoals in de vraag hierboven proberen *Term1* en *Term2* te unificeren. Is dit mogelijk, dan slaagt de afleiding en worden eventuele variabelen gebonden. De (minder belangrijke) tegenhanger van de unificatieoperator is \=. Deze onderzoekt juist of twee termen *niet* unificeerbaar zijn. Deze operator leidt nooit tot een substitutie. Hieronder volgen enige voorbeelden (de laatste drie dienen om er nadrukkelijk op te wijzen dat gebruik van de unificatieoperator *geen numerieke vergelijking* oplevert).

```
| ?-c=c.
yes
| ?-c=d.
no
| ?-c=V.
V=c
yes
| ?-V1=V2,  V2=constante.
V1=constante,
V2=constante
yes
| ?-V1=aap,V1=V2,V2=noot.
no
                    /* Let op: */
| ?-3=2+1.
no
| ?-X+1=3.
no
| ?-X+1\=3.
yes
```

4.2 De anonieme variabele als argument(_)

Als argument in een propositie kan de *anonieme variabele* optreden. Deze bestaat uit *één laagliggend streepje*, _, (een *underscore*). De anonieme variabele is, zoals elke variabele, unificeerbaar met elke willekeurige term. Er is echter één groot verschil: *ieder gebruik van de anonieme variabele is uniek.* Hierdoor kan de anonieme variabele binnen een vraag of clausule met verschillende termen geünificeerd worden. In een al dan niet door de interpretator gerapporteerde substitutie wordt de anonieme variabele niet opgenomen. Andere benamingen voor de anonieme variabele zijn dan ook *don't care* en *dummy* variabele.

Met behulp van de anonieme variabele is soms reductie van het aantal benodigde clausules mogelijk. Bijvoorbeeld in onderstaand woordenboekje

```
koppelwerkwoord(ben,1,enkelvoud).
koppelwerkwoord(bent,2,enkelvoud).
koppelwerkwoord(is,3,enkelvoud).
koppelwerkwoord(zijn,1,meervoud).
koppelwerkwoord(zijn,2,meervoud).
koppelwerkwoord(zijn,3,meervoud).
```

kunnen de laatste drie clausules door één enkele vervangen worden:

```
koppelwerkwoord(ben,1,enkelvoud).
koppelwerkwoord(bent,2,enkelvoud).
koppelwerkwoord(is,3,enkelvoud).
koppelwerkwoord(zijn,_,meervoud).
```

Let wel dat er nooit een constante voor het tweede argument geretourneerd wordt door de clausule met een *streepje* op die plaats. In de volgende vraag krijgt ***P*** geen waarde:

```
| ?-koppelwerkwoord(zijn,P,G).
P=P,
G=meervoud
yes
```

De volgende vraag illustreert het meervoudig voorkomen van de anonieme variabele:

```
| ?-koppelwerkwoord(W,_,_).
W=ben
yes
```

Hierbij ging het de vragensteller kennelijk alleen om een waarde van W, zonder de bijbehorende waarden voor persoon en getal die op de 2e en 3e plaats staan in de **koppelwerkwoord** clausules.

In de volgende *foutieve definitie* van het predikaat **grootouder** wordt getracht de anonieme variabele als tussenschakel te gebruiken:

```
grootouder(G,K):-ouder(_,K),ouder(G,_). /* FOUT */
```

Op de plaatsen van het streepje *moet* een benoemde variabele worden gebruikt, en wel dezelfde, omdat het alleen dan zeker is dat in beide conjuncten dezelfde persoon als *missing-link* functioneert. Alhoewel de tussenpersoon niet in de *oplossing* getoond behoeft te worden, is het voor het verkrijgen daarvan noodzakelijk, dat "intern" een benoemde variabele gebruikt wordt.

Een variabele waarvan de naam uit meer dan één teken bestaat, is níet anoniem, ook al is het eerste teken van de naam een *underscore.* Onderstaande definitie is in orde, omdat een variabele waarvan de naam uit *twee underscrores* bestaat, niet de anonieme variabele is:

```
grootouder(G,K):-ouder(__,K),ouder(G,__).      /* Werkt wel */
```

(Twee *underscores* vormen uiteraard niet de meest sprekende naam!)

Misschien is het goed hier op het volgende te wijzen: *wanneer een benoemde variabele slechts één keer in een clausule voorkomt, is er iets mis*: het bereik van een variabele reikt immers niet verder dan de clausule, dus daarbuiten kan niet naar de variabele verwezen worden. Meestal is er een schrijffout gemaakt en had er een andere variabele, of een constante moeten staan. Is dat niet het geval, dan kan de variabele beter door een anonieme variabele vervangen worden.

4.3 Structuren

4.3.1 Functoren en argumenten

Een *structuur* bestaat uit een *functor met argumenten.* De *functor* bestaat uit een *atoom*; de argumenten staan tussen haakjes achter de functor en bestaan uit een term. Een *n-plaatsige functor* is een functor met *n* argumenten. Onderstaande propositie heeft twee argumenten die beide een *structuur* zijn.

```
predikaat(func1(arg),func2(arg1,arg2))
```

De volgende propositie, die als twee druppels waters op de vorige lijkt, blijkt bij nauwkeurig natellen van de haakjes geheel verschillend daarvan:

```
predikaat(func1(arg,func2(arg1,arg2)))
```

Deze propositie heeft slechts één argument, namelijk een structuur met twee argumenten, waarvan het tweede zelf ook weer een structuur met twee argumenten is. Beide proposities kunnen als argument in een andere propositie opgenomen zijn:

```
super(
        predikaat(func1(arg),func2(arg1,arg2)),
        predikaat(func1(arg,func2(arg1,arg2)))
        )
```

Structuren kunnen uiteraard ook variabelen als argument bevatten.

```
predikaat(func1(V1),func2(V2,V3))
```

Alleen een atoom kan als functor optreden.

```
predikaat(VAR(arg,2(arg2,arg3)))      /* FOUT !!! */
```

VAR en *2* zijn geen atomen, dus bovenstaande rij van symbolen is door de Prologinterpretator niet te verwerken.

4.3.2 Unificatie (versie 2)

In 1.4 zagen we dat twee *proposities* unificeerbaar zijn, indien ze hetzelfde *predikaat* en een gelijk aantal unificeerbare argumenten bezitten. De voorwaarden waaronder structuren unificeerbaar zijn, zijn vrijwel gelijk aan die voor proposities:

> twee *structuren* zijn unificeerbaar wanneer ze dezelfde *functor* hebben en een gelijk aantal unificeerbare argumenten.

De regels voor constanten en variabelen blijven ongewijzigd:

> *gelijke constanten* zijn unificeerbaar;
> *variabelen* zijn unificeerbaar met elkaar;
> een *variabele* is unificeerbaar met een *term* (dus ook met een structuur).

Hieronder volgen enige voorbeelden:

```
| ?-V1=func(a,b,c).
V1=func(a,b,c)
yes
| ?-func(A,b)=func(a,B).
A=a,
B=b
yes
| ?-func(A,b)=func(b,A).
A=b
yes
| ?-func(A,b)=func(A,A).
no
| ?-func(A,b,C)=func(a,B,f(c)).
A= a,
C= f(c),
B=b
yes
| ?-predikaat(func1(a),func2(b,c))=predikaat(F1,F2).
F1=func1(a),
F2=func2(b,c)
yes
| ?-predikaat(func1(a,func2(b,c)))=predikaat(b,c).
no
| ?-predikaat(func1(A,func2(b,C)))=predikaat(func1(a,F2)).
A=a,
C=C,                          /* C is niet gebonden */
F2=func2(b,C)
yes
| ?-predikaat(func1(a,func2(b,c)))=predikaat(F1(a,F2)).
Syntax Error
```

In sommige gevallen komt de interpretator nooit terug met een antwoord, namelijk wanneer een variabele geünificeerd wordt met een structuur die die variabele zelf bevat. In zijn meest duidelijke vorm zien we dat hieronder:

```
| ?-F=oneindig(F).
```

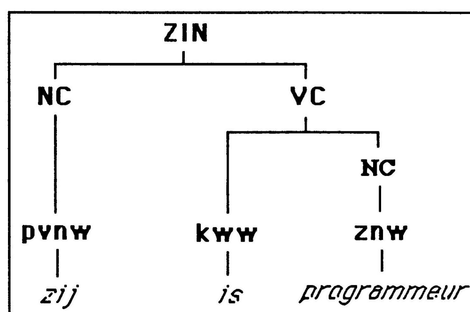
De interpretator wil ***F*** binden aan een structuur en begint met de functor ***oneindig***; die moet nog een argument krijgen en wel ***F***. Welnu, ***F*** is inmiddels gebonden aan een structuur met de functor ***oneindig***, dat wordt alvast ingevuld, nu ***F*** nog even .... De structuur die doorgroeit tot het beschikbare geheugen opgebruikt is, zit als volgt in elkaar

```
oneindig(
     oneindig(
          oneindig(
               oneindig(
```

Er bestaat een Franse Prolog (*Prolog II*, uitgesproken als *deux*), die speciale voorzieningen heeft om met oneindige structuren om te gaan. Deze Prolog, gebaseerd op ideeën van een van de peetvaders van Prolog, Alain Colmerauer, wijkt zowel syntactisch als semantisch op verscheidene punten af van wat op dit moment algemeen gangbaar is. Zie voor deze variant van Prolog, die overigens eerder bestond dan Warrens DEC-10 Edinburgh Prolog, Giannesini e.a. (1986).

4.3.3 Boomstructuren

Met functoren kunnen *boomstructuren* gerepresenteerd worden. Een boomstructuur bestaat uit een aantal objecten die hiërarchisch samenhangen. Voor vele problemen is de boomstructuur een belangrijk *datatype*. We zullen dat illustreren aan de hand van een kleine zinsontleder. Aan alle voorbeeldzinnen uit sectie 3.1 van het vorige hoofdstuk kan dezelfde syntactische structuur toegekend worden als aan het zinnetje hieronder:



Deze *ontleedboom* kan ook beschreven worden met behulp van een *haakjesnotatie*. Hierbij wordt elke knoop in de boom beschreven door zijn *label* met daarachter als argumenten de vertakking(en). De voorbeeldzin bestaat uit een ***nc*** (naamwoordconstituent) plus een ***vc*** (verbale constituent), dus de grove vorm van de structuur is:

zin(nc(...),vc(...))

Maar ***nc*** en ***vc*** zijn elk ook weer vertakt, zodat uiteindelijk de boom hierboven als volgt beschreven kan worden:

zin(nc(pvnw(zij)),vc(kww(is),nc(znw(programmeur))))

De boom is misschien beter te herkennen als we de haakjesstructuur als volgt typen:

```
zin(
        nc(
                pvnw(zij)),
        vc(
                kww(is),
                nc(
                        znw(programmeur))))
```

Het predikaat **boompje**/2 hieronder zoekt bij een drietal woorden de ontleedboom (en omgekeerd):

```
boompje(
        woorden(W1,W2,W3),                            /* 1e argument */
        zin(nc(pvnw(W1)),vc(kww(W2),nc(znw(W3))))    /* 2e argument */
        ) :-
                persoonlijk_voornaamwoord(W1,P,G),
                koppelwerkwoord(W2,P,G),
                zelfstandig_naamwoord(W3,G).
```

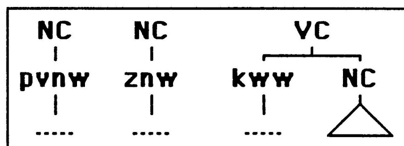
Het predikaat **boompje** heeft slechts twee argumenten (goed de haakjes tellen!). Het eerste bevat de drie woorden, het tweede de boomstructuur. De betekenis van **boompje** is als volgt te omschrijven: met de drie woorden ***W1***, ***W2*** en ***W3*** (respectievelijk een persoonlijk voornaamwoord, een koppelwerkwoord en een zelfstandig naamwoord) en met overeenstemmende ***kenmerken P*** en ***G***, correspondeert een ***zin*** die bestaat uit een ***nc*** en een ***vc***. Beide constituenten hebben ook weer een structuur. Op het laagste niveau bevinden zich de woorden met een aanduiding van hun woordsoort (***pvnw***, ***kww***, ***znw***). Voorbeelden van het gebruik van **boompje** zien we in de volgende sessie:

```
| ?-boompje(woorden(zij,is,programmeur),B).
B=zin(nc(pvnw(zij)),vc(kww(is),nc(znw(programmeur))))
yes
| ?-boompje(woorden(zij,is,programmeur),zin(NC,VC)).
NC=nc(pvnw(zij)),
VC=vc(kww(is),nc(znw(programmeur)))
yes
| ?-boompje(W,_).
W=woorden(ik,ben,programmeur);
W=woorden(ik,ben,psycholoog)
yes
| ?-boompje(woorden(hij,bent,psycholoog),B).
no
```

4.3.4 Opgaven

4.1 Voeg het predikaat **boompje** toe aan de file *zinnetje.pro* en experimenteer ermee. Laat nu eens het ene, dan weer het andere (deel van een) argument variabel zijn.

In de structuur die door **boompje** aan een driewoordszinnetje wordt toegekend, komen de substructuren ***nc*** en ***vc*** voor die als volgt in elkaar steken:



In de volgende opgaven gaan we een predikaat **zin**/2 ontwikkelen, dat hetzelfde doet als **boompje**, maar dat gebruik maakt van de predikaten **nc**/3 en **vc**/3 die substructuren van de zin opleveren.

4.2 Definieer in de file *zinnetje.pro* het *predikaat* **nc**/2, zodat wanneer in een vraag het eerste argument een structuur met de functor ***woorden*** bevat, het tweede argument met de juiste van bovengetekende structuren geünificeerd wordt, en andersom. Bijvoorbeeld:

```
| ?-nc(woorden(programmeur),NC).
NC=nc(znw(programmeur))
yes
| ?-nc(woorden(W),nc(pvnw(W))).
W=ik;
W=jij
yes
```

4.3 Om ervoor te zorgen dat de substructuren van een zin qua ***persoon*** en ***getal*** bij elkaar passen, moeten ook deze substructuren *kenmerken* hebben. Deze kenmerken hangen uiteraard af van de componenten van de substructuur. Voeg aan de definitie van het predikaat **nc** uit de vorige opgave een derde argument toe, bestemd voor een structuur met de tweeplaatsige functor ***kenmerken***. De argumenten van deze functor bestaan uit de ***persoon*** en het ***getal*** die voor de gehele ***nc*** gelden. De kenmerken van een ***nc*** hangen als volgt af van zijn component(en): bestaat de ***nc*** uit een persoonlijk voornaamwoord, dan worden de kenmerken daarvan door de ***nc*** overgenomen; in andere gevallen is de ***persoon*** gelijk aan **3** en wordt het ***getal*** overgenomen van het zelfstandig naamwoord:

```
| ?-nc(woorden(ik),B,K).
B=nc(pvnw(ik)),
K=kenmerken(1,enkelvoud)
yes
| ?-nc(woorden(programmeur),B,kenmerken(P,G)).
B=nc(znw(programmeur)),
P=3,
G=enkelvoud
yes
| ?-nc(woorden(W1),nc(pvnw(W1)),kenmerken(3,enkelvoud)).
W1=hij;
W1=zij;
no
```

4.4 Na definiëring van het predikaat **vc/3**, kunnen we **zin/2** als volgt definiëren:

```
zin(woorden(W1,W2,W3),zin(nc(pvnw(W1)),VC)):-
    nc(woorden(W1),nc(pvnw(W1)),kenmerken(P,G)),
    vc(woorden(W2,W3),VC,kenmerken(P,G)).
```

Het tweede argument van **zin** bevat als argument een structuur met de functor ***zin***. Deze functor heeft twee argumenten. Het eerste daarvan is gelijk aan de ***nc***-structuur die door het predikaat **nc** opgeleverd wordt. Deze structuur is gedeeltelijk al gespecificeerd om er zeker van te zijn dat we op deze plaats alleen een ***nc*** met een persoonlijk voornaamwoord (***pvnw***) krijgen. Zinnen van het volgende type worden dus niet geaccepteerd:

* *Programmeurs zijn wij.*

Het tweede argument is gelijk aan de variabele (*VC*) die door het **vc** predikaat geünificeerd wordt met een *vc*-structuur.
Nu moet het predikaat **vc** nog gedefinieerd worden. Een belangrijke vraag daarbij is, welke kenmerken een ***vc*** moet krijgen, opdat de ***nc*** en de ***vc*** van de ***zin*** dusdanig bij elkaar passen, dat alleen aan grammaticale zinnen een structuur wordt toegekend. Daarvoor beschouwen we de volgende zinnen (achter elk woord staan tussen haakjes de kenmerken aangegeven; zinnen voorzien van een sterretje zijn fout; de zinnen met een vraagteken zijn zeer dubieus, maar die laten we omwille van de opgave toe).

NC	VC	
ik (1,e)	ben (1,e)	programmeur (e)
* ik (1,e)	ben (1,e)	programmeurs (m)
* ik (1,e)	zijn (_,m)	programmeur (e)
wij (1,m)	zijn (_,m)	programmeurs (m)
wij (1,m)	zijn (_,m)	programmeur (e)
? ik (1,e)	ben (1,e)	hij (3,e)
? jij (2,e)	bent (2,e)	zij (2,m)
* wij (1,m)	is (3,e)	jij (2,e)

Uit deze voorbeelden valt te concluderen dat de kenmerken voor een **vc** als volgt beregeld moeten worden, om (voornamelijk) goede zinnetjes te krijgen.

(1) De ***vc*** neemt persoon en getal van het koppelwerkwoord.
(2) Is het getal van de ***vc*** enkelvoud, dan *moet* het getal van de ***nc*** ook enkelvoud zijn.

Definieer op grond van deze informatie het predikaat **vc**.
Bekijk om te testen of alles goed gedefinieerd is onder meer de oplossingen van:

```
| ?-zin(W,Structuur).
```

4.4 Lijsten als argument

Een datatype dat zo mogelijk nog belangrijker is dan een structuur, is de *lijst*. Een *lijst* begint met een rechte haak openen ([) en eindigt met een rechte haak sluiten (]). Tussen de haken staan de *elementen* van de lijst. Elk element is een *term* en kan dus bestaan uit een constante, een variabele, een functor met argumenten én uit een lijst. Het bijzondere aan een lijst is dat het aantal elementen niet vast is: *een lijst heeft geen talligheid*, zoals een functor of een propositie. Hierdoor is het niet bij voorbaat uitgesloten dat lijsten van ongelijke lengte met elkaar geünificeerd worden. Er zijn drie *notatiewijzen* voor het specificeren van de elementen van een lijst, nl. opsomming van alle elementen, specificatie van kop en staart, specificatie van enige beginelementen en de rest.

***Opsomming van de elementen*:** elk element wordt apart aangeduid met een term. Als er geen element in de lijst zit, staat er niets tussen de rechte haken. Een lijst die geen elementen bevat, heet *de lege lijst*. Als er meer dan één element in de lijst zit, worden de elementen van elkaar gescheiden door een komma. Om deze notatiewijze te kunnen gebruiken moet het aantal elementen van de lijst bekend zijn. Voorbeelden van het gebruik van deze notatie zijn:

[]	lege lijst
[een]	1 element: atoom
[2,enkelvoud]	2 elementen: getal en atoom
[P,G]	2 elementen: beide een variabele
[[g,a],[r,u,d,a]]	2 elementen: beide een lijst
[[P,G],vc(v(zijn),nc(n(N)))]	2 elementen: een lijst en een structuur
[[]]	1 element gelijk aan []
[[[M,a,s]]]	1 element gelijk aan [[M,a,s]]
[[][]]	2 elementen: beide de lege lijst

Specificatie van de kop en de staart: de *kop* is het allereerste element van de lijst; de *staart* is de lijst (!) bestaande uit de elementen volgend op de kop. De kop wordt van de staart gescheiden door een rechtopstaande streep | (in het Engels *bar*). De staart is een lijst, hetgeen impliceert dat alle drie de notatiewijzen voor lijsten gebruikt mogen worden om de staart van een lijst te specificeren. Heel belangrijk is dat een staart ook als een variabele gespecificeerd kan worden. Hierdoor kan met deze notatie gerefereerd worden naar elke lijst met meer dan één element. Voorbeelden van lijsten gespecificeerd met behulp van de *kop-en-staart* notatie staan hieronder. In het commentaar staat, waar mogelijk, *dezelfde* lijst beknopter en duidelijker genoteerd door middel van opsomming:

[eerste|Staart]	*kop*=eerste, *staart*=Staart	
[Kop|[e2,e3]]	*kop*=Kop, *staart*=[e2,e3],	*lijst*=**[Kop,e2,e3]**
[kop|[]]	*kop*=kop, *staart*=[],	*lijst*=**[kop]**
[[] | []]	*kop*=[], *staart*=[],	*lijst*=**[[]]**
[1|[2 | Staart_van_staart]]	*kop*=1, *staart*=[2|Staart_van_staart]	
[premier|_]	*kop*=premier, *staart*=_	

Specificatie van enige beginelementen en de rest. Dit is een combinatie van de twee vorige beschrijvingsmethoden: enige beginelementen gescheiden door komma's worden expliciet gespecificeerd en door een rechte streep gescheiden van de overige elementen. Ook hier behoeft het aantal elementen niet bekend te zijn. Wordt deze notatie gebruikt, dan wordt er van uitgegaan dat de lengte van de lijst minimaal gelijk is aan het aantal expliciet aangegeven beginelementen).
In de volgende voorbeelden zijn, waar mogelijk, beknoptere schrijfwijzen aangegeven:

```
[e1,e2,e3|Staart]
[Head,Second|_]
[[boter,kaas,eieren],branche(zuivel)| Overige]
[1|[2 | Staart_van_staart]]                [1,2,Staart_van staart]
[1|[2|[3,4|T]]]                            [1,2,3,4|T]
[1,2|[_,4]]                                [1,2,_,4]
```

Hieronder volgt een lijst waarvan de vijf beginelementen tussen apostrophen staande atomen zijn (een komma, een puntkomma, een], een [en een |):

```
[',',';',']','[','|'|Andere_atomen]        5 elementen + staart
```

Voor alle duidelijkheid volgen hier nog enige syntactische eigenschappen van lijsten.

Vóór de rechte streep staat altijd minstens één element.
Fout is: [|T].

Na de rechte streep staat altijd precies één term, die unificeerbaar is met een *lijst*.
Fout zijn: [e1|e2], [e1|e2,e3]

Direct naast de rechte streep staat nooit een komma.
Fout zijn: [e1,|[e2,e3]], [e1|,[e2,e3]], [a,|,b]

Een komma kan uitsluitend twee expliciet opgegeven elementen scheiden.
Fout zijn: [1,,3,4], [,]

Met de *kop* van een lijst duiden we het eerste element van die lijst aan; onder de *staart* van een lijst verstaan we de lijst bestaande uit de elementen ná de kop (de Engelse termen zijn *head* en *tail*). Deze aanduidingen gelden ongeacht de manier waarop een lijst genoteerd staat.
Twee lijsten zijn unificeerbaar wanneer de kop en de staart unificeerbaar zijn. Dit zien we in de volgende sessie gedemonstreerd met behulp van de unificatieoperator:

```
| ?-[a,B,c]=[a,b,C].
B=b,
C=c
yes
| ?-[a,b,c]=[a,b|C].
C=[c]
yes
| ?-[a,b,c]=[a,b|[C]].
C=c
yes
| ?-[a,b,c]=[a,b,c|T].
T=[]
yes
| ?-[H|[b,c]]=[a,b,c].
H=a
yes
| ?-[a|[b|[c]]]=[a,b,c].            /* ofwel: [a,b,c]=[a,b,c]. */
yes
| ?-[a,b|T]=[a,b|U].
T=U
yes
```

4.4.1 Opgaven

4.5 Onderzoek welk van de volgende expressies lijst zijn en welke niet. Als een expressie een lijst is, som dan voor zover mogelijk de elementen op. Als niet alle elementen opgesomd kunnen worden, geef dan aan wat de staart is.

(1) **(a,b,c)**
(2) **[]**
(3) **A**
(4) **[a,b]**
(5) **[a,b|c]**
(6) **[a,b|[c]]**
(7) **[a,b,[c]]**
(8) **[a,b,|,[c]]**
(9) **[|T]**
(10) **[,|T]**
(11) **[a,b|T]**
(12) **[f(a),g(b,c])**
(13) **lijst([a,b])**
(14) **[lijst([a,b])]**

4.6 In de file *zinnetje.pro* worden de woorden en de kenmerken gespecificeerd door speciale functoren te gebruiken. Vervang deze structuren overal door lijsten. Vervang in alle definities bijvoorbeeld de structuur

woorden(W1,W2,W3)

door de lijst

[W1,W2,W3]

In het woordenboekje staan de kenmerken van de woorden als losse argumenten. Zet voor de uniformiteit ook deze in een lijstje. Vervang bijvoorbeeld het drieplaatsige feit

persoonlijk_voornaamwoord(ik,1,enkelvoud).

door het tweeplaatsige:

persoonlijk_voornaamwoord(ik,[1,enkelvoud]).

Test na alle wijzigingen of het programma nog loopt. Probeer ook eens:

| ?-**zin([hij|Rest],Structuur).**

4.5 Unificatie (definitieve versie)

In deze sectie zetten we alles aangaande termen en unificatie van termen nog eens op een rijtje. We beginnen met het begrip *term*:

> Een *constante* (een getal of atoom) is een term.
>
> Een *variabele* is een term.
>
> Als *f* een atoom is en $t_1, t_2, ..., t_n$ zijn termen, dan is ook de n-plaatsige *structuur* $f(t_1, t_2, ..., t_n)$ een term[1].
>
> *[]* is een *lijst*. Als $t_1, t_2, ..., t_n$ termen zijn, dan is $[t_1, t_2, ..., t_n]$ een lijst. Is *L* een lijst, dan is ook $[t_1, t_2, ...|L]$ een lijst. Ook een lijst is een term.

Om te beschrijven wanneer twee termen unificeerbaar zijn, gaan we de mogelijkheden af.

> Twee *constanten* zijn unificeerbaar als ze gelijk zijn.
>
> Een *variabele V* is unificeerbaar met iedere willekeurige term *T* door de *substitutie V=T*.
>
> Twee *structuren* $f(t_1, t_2, ..., t_m)$ en $g(v_1, v_2, ..., v_n)$ zijn unificeerbaar, indien *f* gelijk is aan *g*, *m* gelijk is aan *n* en de qua plaats corresponderende argumenten unificeerbaar zijn.
>
> Twee *lijsten [H1|T1]* en *[H2|T2]* zijn unificeerbaar, indien ***H1*** met ***H2***, en ***T1*** met ***T2*** unificeerbaar is.

4.5.1 Opgaven

4.7 Probeer op de computer uit wat het resultaat is van de volgende toepassingen van de unificatieoperator. Het is verstandig te trachten de resultaten te voorspellen (soms staan de haakjes met opzet verkeerd):

	Voorspelde substitutie:			
[a,b,c]=[X,Y|Z].	X=	Y=	Z=	
[a,b,c]=[X,Y,Z|T].	X=	Y=	Z=	T=
[a,b,c]=[X,Y,Z,V|T].	X=	Y=	Z=	T=
[a,b,c]=[X|Y],Y=[Z|T].	X=	Y=	Z=	T=
[a,b,c]=[X,X,X].	X=			
[a,a,a]=[X,X,X].	X=			
[a,a,a]=[X,Y,Z].	X=	Y=	Z=	
[a,a,x]=[X,Y,Z],X=Y.	X=	Y=	Z=	
[[a,b],[c,d]]=[X,Y].	X=	Y=		
[[a,b],[c,d]]=[X|Y].	X=	Y=		
func([a,b],2)=func(X,2).	X=			
[func([a,b]),func([p,q])]=[Y,Z].		Y=	Z=	
func(f(g(a)),k(g(a)))=func(Z,k(T(a))).			Z=	T=

[1] *De drie puntjes (...) vormen geen letterlijk onderdeel van de term!*

4.8 Vaak is het nodig uit een lijst één element of een sublijst te selecteren. In onderstaande sessie worden enige predikaten getoond. Elk van de predikaten heeft als eerste argument een lijst, en als tweede argument een selectie daaruit. De namen van de predikaten geven (hopelijk) weer wat geselecteerd wordt.

```
| ?-kop([a,b,c],K).
K=a
yes
| ?-kop([],K).
no
| ?-staart([a,b,c],S).
S=[b,c]
yes
| ?-staart([a],S).
S=[]
yes
| ?-staart([],S).
no
| ?-tweede([a,b,c],E).
E=b
yes
| ?-na_tweede([a,b,c],R).
R=[c]
yes
| ?-eerste_twee([a,b,c],E1E2).
E1E2=[a,b]
yes
| ?-kop_van_kop([[a1,a2,a3],b,c],KvK).
KvK=a1
yes
| ?-staart_van_kop([[a1,a2,a3],b,c],SvK).
SvK=[a2,a3]
yes
```

Door goed de verschillende notatiewijzen voor lijsten toe te passen, is het mogelijk alle hierboven gebruikte predikaten te definiëren als *feit*. (Zet de definities in de file *lisp.pro*.) Bij wijze van voorbeeld staan hieronder de *definities* van een paar predikaten van hetzelfde type als bovenstaande:

```
derde([_,_,X|_],X).
na_derde([_,_,_|Rest],Rest).
```

Bovenstaande predikaten en de nog te definiëren predikaten "doen" iets, zonder dat daar een reductiestap voor nodig is. Een unificatiestap is voldoende. Zuiniger definities zijn niet mogelijk. De definitie

```
na_derde([A|[B|[C|Rest]]],X):-X=Rest.          /* BOMBASTISCH */
```

is weliswaar correct, maar minder efficiënt vanwege vier overbodige variabelen, de expliciete unificatie en de vereiste reductiestap. Bovendien is de lijst onnodig ingewikkeld genoteerd. Definieer in de file *lisp.pro* de predikaten die in de voorbeeldsessie gedemonstreerd werden.

5 Recursie

Een belangrijke programmeertechniek is het *recursief definiëren van predikaten*. De definitie van een predikaat is *recursief* als dat predikaat (al of niet via andere predikaten) gebruikt wordt in de condities van de clausules die dat predikaat definiëren. Met andere woorden: het predikaat wordt in zijn definitie (gedeeltelijk) in zichzelf uitgedrukt. **r** in het volgende voorbeeld is recursief:

```
r:-q.
r:-s,r.
```

Een ander voorbeeld, waarbij twee predikaten **r1** en **r2** via elkaar recursief zijn, is het volgende:

```
r1.
r1:-x,r2.

r2:-y.
r2:-r1,z.
```

In recursieve definities zijn altijd te onderscheiden één of meer *stopclausules* en één of meer *reducerende clausules*. Een *stopclausule* beëindigt de afleiding hetzij omdat er een oplossing gevonden is, hetzij omdat er geen oplossing meer mogelijk is. Een afleiding kan ook beëindigd worden, doordat geen enkele clausule (meer) van toepassing is. Een *reducerende clausule* brengt het probleem terug tot een probleem van hetzelfde type, maar dan *van een kleinere orde*. Vaak wordt in een reducerende clausule een deel van de oplossing geconstrueerd. Een predikaat dat in zichzelf wordt uitgedrukt zonder dat er een reductie van het probleem plaats vindt zal nooit een oplossing opleveren, maar eeuwig doorlopen, of het beschikbare geheugen opgebruiken (veelal met de melding *stack overflow*). We zullen enige voorbeelden van recursieve programma's bespreken. Recepten zijn niet te geven, hoogstens wat vuistregels.

5.1 Recursie en lijsten

5.1.1 is_elem_van

Bij *lijsten* fungeert de recursie meestal om de elementen van een lijst één voor één af te werken. Hierbij wordt de lijst gesplitst in de kop en de staart. Het volgende predikaat is daar een voorbeeld van:

```
is_elem_van(X,[X|_]).
is_elem_van(X,[_|T]):-is_elem_van(X,T).
```

Het predikaat **is_elem_van** is recursief gedefinieerd, omdat in de tweede clausule van de definitie ditzelfde predikaat rechts van het indien-teken voorkomt. Dit belangrijke predikaat (in het Engels *member* geheten) zoekt uit of het eerste argument (*X*) voorkomt onder de elementen

van de lijst in het tweede argument. Hieronder zien we een succesvolle en een niet-succesvolle toepassing van **is_elem_van**:

```
| ?-is_elem_van(b,[a,b,c]).
yes
| ?-is_elem_van(d,[a,b,c]).
no
```

De logica achter de clausules blijkt als we het programmaatje hierboven "vertalen" naar het Nederlands (*X* is het gezochte element en *L* is de te doorzoeken lijst):

> ***X*** is een element van de ***niet lege lijst L***:
> als ***X*** unificeerbaar is met de kop van ***L*** (*stopconditie*), *of*
> als ***X*** een element van de staart van ***L*** is (*reducerende stap*).

Om te bepalen of een element ***X*** in een lijst ***L*** voorkomt, wordt eerst onderzocht of ***X*** unificeerbaar is met de kop van ***L***. Is dat het geval, dan is de afleiding succesvol. Is dat niet het geval, dan wordt het probleem gereduceerd tot een probleem met een lijst die één element minder heeft dan de oorspronkelijke lijst (namelijk de staart). Deze reductie gaat door tot het element gevonden is, óf totdat de te reduceren lijst leeg is, in welk geval de afleiding mislukt is. Dat de afleiding stopt op het moment dat de lege lijst bereikt is, is als volgt te begrijpen: in beide clausules van **is_elem_van** zijn kop en staart van de te onderzoeken lijst expliciet aangegeven. In de eerste clausule staat de lijst genoteerd als *[X|_]*, in de tweede als *[_|T]*. Dit impliceert dat beide clausules alleen toepasbaar zijn als er nog minstens één element over is.
Hoe de af te zoeken lijst steeds korter wordt, zien we in de volgende sessie waarin we de *tracer* gebruiken om te kunnen volgen hoe de interpretator tot een bevestigend of ontkennend antwoord komt:

```
||?-is_elem_van(b,[a,b,c]).
[1] 0 Call: is_elem_van(b,[a,b,c]) ?
Match: is_elem_van(b,[a,b,c]):-is_elem_van(b,[b,c]).
        [2] 1 Call: is_elem_van(b,[b,c]) ?
        Match : is_elem_van(b,[b,c]).
        [2] 1 Exit: is_elem_van(b,[b,c])
[1] 0 Exit: is_elem_van(b,[a,b,c])
yes

||?-is_elem_van(d,[a,b,c]).
[1] 0 Call: is_elem_van(d,[a,b,c]) ?
Match : is_elem_van(d,[a,b,c]):-is_elem_van(d,[b,c]).
        [2] 1 Call: is_elem_van(d,[b,c]) ?
        Match : is_elem_van(d,[b,c]):-is_elem_van(d,[c]).
                [3] 2 Call: is_elem_van(d,[c]) ?
                Match: is_elem_van(d,[c]):-is_elem_van(d,[]).
                        [4] 3 Call: is_elem_van(d,[]) ?
                        [4] 3 Fail: is_elem_van(d,[])
                [3] 2 Pop: is_elem_van(d,[c])
        [2] 1 Pop: is_elem_van(d,[b,c])
[1] 0 Pop: is_elem_van(d,[a,b,c])
no
```

Uit bovenstaande *traces* blijkt hoe eenzelfde clausule op verschillende niveaus in een afleiding kan functioneren. Dit is mogelijk, doordat de interpretator, zoals we reeds in hoofdstuk 1

opgemerkt hebben, altijd een kopie van een clausule (feit of regel) gebruikt om een afleiding te construeren.

5.1.2 append/3

Als tweede voorbeeld van een recusief predikaat definiëren we opnieuw een predikaat dat voor het werken met lijsten erg belangrijk is, **append/3**, dat bedoeld is om twee lijsten aan elkaar te koppelen, maar dat bij nader onderzoek ook in staat blijkt een lijst op te splisten.

5.1.2.1 append als samenvoeger

Met **append** kunnen lijsten samengevoegd worden, waarvan hieronder een voorbeeld:

```
| ?-append([een,twee,drie],[4,5,6,7],SamenGevoegdeL).
SamenGevoegdeL=[een,twee,drie,4,5,6,7]
yes
```

Om **append** te definiëren zoeken we eerst naar *het triviale, direct oplosbare geval* dat als eindconditie kan gelden, en waarnaar elk appendprobleem moet worden teruggebracht. Dit triviale geval is het volgende:

> Is de eerste lijst *leeg*, dan is de samengevoegde lijst gelijk aan de tweede lijst

Hebben we niet te maken met het triviale geval, dan reduceren we ons probleem een stapje in die richting:

> Is de eerste lijst niet leeg, dan is de samengevoegde lijst gelijk aan de kop van de eerste lijst, met daarachter de append van de staart van de eerste lijst met de tweede lijst.

Aangezien de staart van de eerste lijst een element korter is dan de oorspronkelijke lijst hebben we een stap in de goede richting gezet. Het aldus ontstane appendprobleem reduceert het probleem nogmaals. Op een gegeven moment wordt het triviale geval bereikt en opgelost. Daarmee is dankzij een *keten van substituties* het hele probleem opgelost. In Prolog zien de stopconditie en de reducerende stap er als volgt uit:

```
append([],L2,L2).
append([KopvanL1|StaartvanL1],L2,[KopvanL1|StaartvanL3]):-
        append(StaartvanL1,L2,StaartvanL3).
```

De stapsgewijze reductie van de eerste lijst en het vervolgens groeien van de derde lijst zien we duidelijk in de trace op de volgende bladzijde.

We zien in de tweede clausule van de definitie van **append** weer een voorbeeld van de kracht van de unificatiestap: de toevoeging van de kop van de eerste lijst (***KopvanL1***) aan het eindresultaat geschiedt al op het moment dat het hoofd van die regel geünificeerd wordt met de doelpropositie die die clausule activeert. Op dat moment wordt ***KopvanL1*** geünificeerd met het eerste element van de eerste lijst; tegelijkertijd wordt het derde argument geünificeerd met een lijst, waarvan de staart weliswaar nog niet gebonden is (***StaartvanL3***), maar waarvan het eerste element geünificeerd wordt met ***KopvanL1***. Wat de staart van het eindresultaat (***StaartvanL3***) zal zijn, wordt aan de rechterkant van de regel beschreven als de **append** van de staart van de eerste lijst (***StaartvanL1***) en de ongewijzigde tweede lijst (***Lijst2***). De eerste clausule zorgt ervoor dat na de nodige reducerende stappen de staart van het eindresultaat (***StaartvanL3_3***) geünificeerd wordt met ***L2***. De tracer heeft ten behoeve van de leesbaarheid

aan de ongebonden variabele ***StaartvanL3*** overal een nummer gelijk aan de diepte van de afleiding toegevoegd, hetgeen volkomen terecht is, daar de variabelen op de verschillende niveaus in feite ook verschillende variabelen zíjn. Het bereik van een variabele is immers de clausule, en clausules op verschillende diepte in een afleiding vormen elk een eigen bereik voor de variabelen in die clausule. De verschillende clausules zijn wel aan elkaar gerelateerd via substituties.

```
||?-append([een,twee,drie],[4,5,6,7],SamenGevoegdeL).
[1] 0 Call: append([een,twee,drie],[4,5,6,7],SamenGevoegdeL)?
Match: append([een,twee,drie],[4,5,6,7],[een|StaartvanL3_1]):-
            append([twee,drie],[4,5,6,7],StaartvanL3_1).
        [2] 1 Call: append([twee,drie],[4,5,6,7],StaartvanL3_1)?
        Match: append([twee,drie],[4,5,6,7],[twee|StaartvanL3_2]):-
                    append([drie],[4,5,6,7],StaartvanL3_2).
                [3] 2 Call: append([drie],[4,5,6,7],StaartvanL3_2)?
                Match: append([drie],[4,5,6,7],[drie|StaartvanL3_3]):-
                            append([],[4,5,6,7],StaartvanL3_3).
                        [4] 3 Call: append([],[4,5,6,7],StaartvanL3_3)?
                        Match: append([],[4,5,6,7],[4,5,6,7]).
                        [4] 3 Exit: append([],[4,5,6,7],[4,5,6,7])
                [3] 2 Exit: append([drie],[4,5,6,7],[drie,4,5,6,7])
        [2] 1 Exit: append([twee,drie],[4,5,6,7],[twee,drie,4,5,6,7])
[1] 0 Exit: append([een,twee,drie],[4,5,6,7],[een,twee,drie,4,5,6,7])
SamenGevoegdeL=[een,twee,drie,4,5,6,7]
yes
```

Bij toepassing van **append** moet niet vergeten worden dat uitsluitend *lijsten* aan elkaar gekoppeld kunnen worden. Wanneer bijvoorbeeld ***E*** als *element* aan het einde van een lijst moet worden toegevoegd, dan moet ***E*** eerst "ingepakt" worden. Het moet dus níet zo:

```
| ?-E=c,  append([a,b],E,X).          /* FOUT !! */
```

maar zó:

```
| ?-E=c,  append([a,b],[E],X).        /* GOED */
L=[a,b,c]
yes
```

Als ***E*** een lijst zou zijn, dan worden in het eerste voorbeeld de lijst ***[a,b]*** en ***E*** weliswaar aan elkaar gekoppeld, maar we wilden ***E*** als *element* in de lijst. Vergelijk:

```
| ?-E=[c,d],  append([a,b],E,L).
L=[a,b,c,d]
yes
```

en

```
| ?-E=[c,d],  append([a,b],[E],L).
L=[a,b,[c,d]]
yes
```

In het laatste voorbeeld zit de lijst ***[c,d]*** in zijn geheel als element in ***L***.

5.1.2.2 append als splitser

Unificatie werkt twee kanten op. De interpretator kan daardoor ook een doelpropositie aan die bestaat uit een **append**, waarbij het *derde* argument gebonden is en de overige twee variabel zijn.

```
||?-append(EersteDeel,TweedeDeel,[a,e,i,o,u,y]).
EersteDeel=[],
TweedeDeel=[a,e,i,o,u,y];          /* puntkomma */
EersteDeel=[a],
TweedeDeel=[e,i,o,u,y];            /* puntkomma */
EersteDeel=[a,e],
TweedeDeel=[i,o,u,y]               /* zo is het wel genoeg */
yes
```

In dit geval wordt de variabele ***KopvanL1*** gebonden door de kop van het derde argument en niet door de kop van het eerste argument (zie de trace hiervoor). Er "verhuist" hier dus een element van het laatste element naar het eerste. Ook hier fungeert de eerste clausule als eindconditie. Ter bestudering in de leunstoel volgt hier een trace:

```
||?-append(EersteDeel,TweedeDeel,[a,e,i,o,u,y]).
[1] 0 Call: append(EersteDeel,TweedeDeel,[a,e,i,o,u,y])?
Match: append([],[a,e,i,o,u,y],[a,e,i,o,u,y]).
[1] 0 Exit: append([],[a,e,i,o,u,y],[a,e,i,o,u,y])
EersteDeel=[],
TweedeDeel=[a,e,i,o,u,y];
[1] 0 Pop: append([],[a,e,i,o,u,y],[a,e,i,o,u,y])
[1] 0 Redo: append(EersteDeel,TweedeDeel,[a,e,i,o,u,y])?
Match: append([a|StaartvanL1_1],TweedeDeel,[a,e,i,o,u,y]):-
                append(StaartvanL1_1,TweedeDeel,[e,i,o,u,y]).
    [2] 1 Call: append(StaartvanL1_1,TweedeDeel,[e,i,o,u,y])?
    Match: append([],[e,i,o,u,y],[e,i,o,u,y]).
    [2] 1 Exit: append([],[e,i,o,u,y],[e,i,o,u,y])
[1] 0 Exit: append([a],[e,i,o,u,y],[a,e,i,o,u,y])
EersteDeel=[a],
TweedeDeel=[e,i,o,u,y];
    [2] 1 Pop: append([],[e,i,o,u,y],[e,i,o,u,y])
    [2] 1 Redo: append(StaartvanL1_1,TweedeDeel,[e,i,o,u,y])?
    Match: append([e|StaartvanL1_2],TweedeDeel,[e,i,o,u,y]):-
                    append(StaartvanL1_2,TweedeDeel,[i,o,u,y]).
        [3] 2 Call: append(StaartvanL1_2,TweedeDeel,[i,o,u,y])?
        Match: append([],[i,o,u,y],[i,o,u,y]).
        [3] 2 Exit: append([],[i,o,u,y],[i,o,u,y])
    [2] 1 Exit: append([e],[i,o,u,y],[e,i,o,u,y])
[1] 0 Exit: append([a,e],[i,o,u,y],[a,e,i,o,u,y])
EersteDeel=[a,e],
TweedeDeel=[i,o,u,y]
yes
```

Er zijn nog andere gebruiksmogelijkheden voor **append** (zie o.a. opgave ***5.2***).

5.1.3 Reverse/2

We definiëren een derde belangrijk predikaat, **reverse/2**, dat de elementen van een lijst in omgekeerde volgorde zet:

```
| ?-reverse([a,b,c],L).
L=[c,b,a]
yes
```

We beschrijven eerst weer de stappen van een versie van **reverse** waar op het eerste gezicht niets aan lijkt te mankeren (de zogenaamde *naïeve reverse*).

> Hoe berekenen we de reverse R van een lijst L?
> *(stopclausule)* In het triviale geval is L de lege lijst en dan is R ook leeg, anders is *(reducerende stap)* R gelijk aan de reverse van de staart van L, met daarachter de kop van L.

Ook hier zien we weer dat het probleem in de tweede stap wordt gereduceerd tot een probleem met een lijst die een element minder heeft. De reductie gaat door tot we bij de lege lijst aangeland zijn. Vertaald in Prolog krijgen we:

```
naieve_reverse([],[]).
naieve_reverse([H|T],R):-
        naieve_reverse(T,RevT),
        append(RevT,[H],R).
```

Dit programma heeft als nadeel dat vele lijsten doorlopen worden, waardoor deze definitie erg inefficiënt is. De hele lijst wordt een keer doorlopen door de **naieve_reverse** zelf. Bovendien leidt elke reducerende stap tot een toepassing van **append** waarbij de staart van de lijst doorlopen moet worden. Is de om te keren lijst 10 lang, dan moeten dus lijsten van 10, 9, 8 enzovoorts doorlopen worden, hetgeen tot een indrukwekkende trace leidt. Als wij zelf de volgorde van een stapel floppies moeten omkeren, dan doen we dat door de bovenste floppy van de stapel te halen, die op tafel te leggen, en vervolgens de volgende daar bovenop te leggen enzovoorts. Een dergelijke procedure wordt in de volgende definitie toegepast, door middel van een *extra argument voor tussenresultaten* (*Stapel*):

```
reverse(L,R):-reverse(L,[],R).

reverse([],Stapel,Stapel).
reverse([Kop|Staart],Stapel,R):-
        reverse(Staart,[Kop|Stapel],R).
```

Hierboven wordt **reverse/2** gedefinieerd met behulp van **reverse/3**, dat het echte werk doet. Het tweede argument van **reverse/3** fungeert als de stapel. De startwaarde van de stapel is de lege lijst. Zolang de lijst in het eerste argument niet leeg is, reduceert **reverse/3** zich tot een probleem waarbij de om te keren lijst teruggebracht wordt tot zijn staart, maar waarbij de stapel groeit door toevoeging van de kop van de lijst (*[Kop|Stapel]*). Het derde argument (***R***) dat bestemd is voor de omgekeerde lijst, wordt zonder wijziging meegenomen de diepte in, totdat de om te keren lijst leeg is. Dan wordt de **Stapel,** die op dat moment de elementen van de leeggehaalde lijst in omgekeerde volgorde bevat, door de stopclausule van **reverse/3** geünificeerd met het derde argument van **reverse/3**. En dit argument wordt op zijn beurt geünificeerd met het tweede argument van **reverse** (***R***). In deze versie van **reverse** wordt de

lijst slechts één keer doorlopen. De naïeve versie produceert dermate veel reductiestappen, dat die versie vaak gebruikt wordt als test om de snelheid van een Prologimplementatie te bepalen.

5.1.4 Opgaven

In het algemeen is het nuttig om te onderzoeken wat er gebeurt wanneer een predikaat voor iets anders gebruikt wordt dan waarvoor het in eerste instantie bedoeld is, bijvoorbeeld door in de argumenten meer of minder variabelen op te nemen of door variabelen op andere plaatsen te zetten dan normaal. Ook een goede gewoonte bij het uittesten van predikaten is na te gaan wat er gebeurt wanneer er na de eerste, meestal gewenste, oplossing nog een oplossing gevraagd wordt.
We zagen hier al voorbeelden van. Hieronder volgen er nog enige.

5.1 Zet de definitie van **is_elem_van**/2 in de file *lisp.pro*.
(a) Ga met behulp van de tracer na wat er gebeurt als de eerste clausule ontbreekt.
(b) Werkt **is_elem_van** ook als de clausules in omgekeerde volgorde staan? Gebruik de tracer om te zien wat er gebeurt.
(c) Wat gebeurt er als in een doelpropositie het eerste argument een variabele is? Dit gedrag van **is_elem_van**, dat een gevolg is van de symmetrie van unificatie, kan van pas komen wanneer de elementen van een lijst om de beurt ergens voor nodig zijn.
(d) Wat gebeurt er als het tweede argument een variabele is?
(e) En wat als beide argumenten variabel zijn?

5.2 Voeg de definitie van **append** toe aan de file *lisp.pro*. Probeer de voorbeelden uit de tekst. Onderzoek met behulp van de tracer hoe de volgende vraag verwerkt wordt:

```
| ?-append([Hoeveel],[' ',km],[12,' ',km]).
Hoeveel=12
yes
```

5.3 Definieer het predikaat **laatste**/2 (ook in *lisp.pro*) dat van een lijst in het eerste argument het laatste element unificeert met het tweede argument. Definieer het predikaat zodanig dat het slechts één oplossing heeft. Dit kan door ervoor te zorgen dat de reducerende clausule slechts toepasbaar is wanneer een lijst minstens twee elementen heeft. Een voorbeeld:

```
| ?-laatste([a,b,[c]],Laatste).
Laatste=[c];
no
| ?-laatste([],Laatste).
no
```

5.4 (a) Vergelijk de traces van beide versies van ***reverse*** toegepast op eenzelfde lijst.
(b) Men zou in een aanval van imperatief programmeren (zie hoofdstuk 0) kunnen denken dat de derde variabele van **reverse**/3 overbodig is, omdat het gewenste resultaat tóch al in het tweede argument staat. Ga o.a. met de tracer de rol van de derde variabele na. Laat hem eens weg en overzie de gevolgen.

5.5 (a) Schrijf, in de file *lisp.pro,* een recursief predikaat **voor**/3, dat nagaat of het eerste argument vóór het tweede argument voorkomt in het derde argument, dat een lijst bevat. (Hierbij mag er van uitgegaan worden dat elk element maar één keer in de lijst voorkomt.) Dus:

```
| ?-voor(aap,mies,[aap,noot,mies,truus]).
yes
```

```
| ?-voor(noot,aap,[aap,noot,mies,truus]).
no
```

We geven hier de stappen van de oplossing:

> Als de kop van de lijst gelijk is aan het 1e argument, onderzoek dan of het 2e argument een element van de staart van de lijst is. (*Einde van de reductie*).
> Pas **voor** toe op de staart van de lijst (*reducerende stap*).

(b) Definieer met behulp van **voor/3** het complementaire predikaat **na/3**.
(c) Vaak is het nodig om aan niet numerieke concepten een rangorde toe te kennen. Zo zijn bijvoorbeeld woorden alfabetisch geordend, kleuren meer of minder donker en programmeertalen meer of minder mooi. In al dit soort gevallen kan **voor** gebruikt worden om de ordening te definiëren. Dat is ook het geval in het volgende probleem (dat niet bedoeld is als illustratie van recursief programmeren!).
Definieer in de file *kaarten.pro* een predikaat **kaart**/2 dat een kaartspel representeert. Het eerste argument bevat de kleur (schoppen, harten, ruiten, klaveren), het tweede de waarde (aas, heer, ..., 3, 2). Tip: definieer hulppredikaten **kleur**/1 en **waarde**/1, zodat niet 52 clausules geschreven behoeven te worden, maar slechts 18 (1 regel en 4+13 feiten). Gebruik **voor** om de kleuren van twee kaarten te vergelijken. De rangorde in afnemende volgorde moet zijn: schoppen, harten, ruiten en klaveren. Dus:

```
| ?-hogere_kleur(kaart(ruiten,2),kaart(klaveren,aas)).
yes
```

5.6 Gebruik de twee verschillende gebruiksmogelijkheden van **append** (koppelen en splitsen) om (niet recursief) een predikaat **pick**/3 te definiëren dat telkens één element uit de lijst van het eerste argument in het tweede argument zet, en de overige elementen van die lijst in het derde argument.

```
| ?-pick([a,b,c,d],E,R).
E=a
R=[b,c,d];                    /* <--- puntkomma ingetypt */
E=b
R=[a,c,d];                    /* <--- puntkomma ingetypt */
E=c
R=[a,b,d]                     /* <--- return!*/
yes
```

(Eén regel met rechts een conjunctie van twee toepassingen van **append** is voldoende. De kunst is om, zoals zo vaak, de lijsten goed te noteren.)

5.7 Definieer recursief een predikaat **permutatie**/2 dat alle mogelijke volgordes van de elementen van een lijst doorloopt:

```
| ?-permutatie([a,b,c],P).
P=[a,b,c];                    /* <--- puntkomma ingetypt */
P=[a,c,b];                    /* <--- puntkomma ingetypt */
P=[b,a,c];                    /* <--- puntkomma ingetypt */
(enzovoorts)
```

Aanwijzing: Breng het probleem voor een lijst terug naar een lijst met een element minder door

middel van **pick/3**. Permuteer de gereduceerde lijst en zet het eruit genomen element voor deze permutatie. Dit moet uiteraard voor elk element van een lijst gebeuren.

5.8 Definieer het tweeplaatsige predikaat **match**, dat onderzoekt of een lijst "matcht" met een op te geven patroon.

> Het eerste argument bevat het *patroon* (een lijst).
> Het tweede argument bevat de te onderzoeken lijst.

Als we het patroon met ***P*** en de lijst met ***L*** aangeven, is de vorm:

match(P,L).

In het eenvoudigste geval matcht ***L*** met ***P***, als ***L*** evenveel elementen bevat als ***P*** en elk element van ***L*** gelijk is aan het corresponderende element in ***P***. (We gaan er in deze opgave van uit dat noch ***P***, noch ***L*** variabelen bevat.)

Voorbeelden:

```
| ?-match([piet,is,de,broer,van,jan],[piet,is,de,broer,van,jan]).
yes
| ?-match([piet,is,de,neef,van,jan],[piet,is,de,broer,van,jan]).
no
| ?-match([piet,is,de,broer],[piet,is,de,broer,van,jan]).
no
| ?-match([piet,is,de,broer,van,jan],[piet,is,de,broer]).
no
```

In het patroon kunnen behalve gewone elementen drie *speciale symbolen* voorkomen, nl. een vraagteken, een plusteken en een sterretje, welke de volgende betekenis hebben:

? in ***P***:	matcht met *precies 1 element* in ***L***
+ in ***P:***	matcht met *1 of meer elementen* in ***L***
* in ***P:***	matcht met *0 of meer elementen* in ***L***

Voorbeelden van het gebruik van de speciale symbolen zijn:

```
| ?-match([piet,?,?,broer,?,jan],[piet,is,de,broer,van,jan]).
yes
| ?-match([piet,?,broer,?,jan],[piet,is,de,broer,van,jan]).
no
| ?-match([piet,+,broer,?,jan],[piet,is,de,broer,van,jan]).
yes
| ?-match([+,broer,+],[piet,is,de,broer,van,jan]).
yes
| ?-match([piet,is,de,broer,van,jan,+],[piet,is,de,broer,van,jan]).
no
| ?-match([piet,is,de,+,broer,van,jan],[piet,is,de,broer,van,jan]).
no
| ?-match([piet,is,de,broer,van,jan,*],[piet,is,de,broer,van,jan]).
yes
```

```
| ?-match([piet,is,de,*,broer,van,jan],[piet,is,de,broer,van,jan]).
yes
| ?-match([*,broer,*],[piet,is,de,broer,van,jan]).
yes
```

P en ***L*** mogen *lijsten als element* hebben. Deze behoeven zelf *niet* ontleed te worden: een lijst in ***P*** moet eenvoudigweg unificeerbaar zijn met een corresponderend element uit ***L***, wil er een match mogelijk zijn. (Een gevolg hiervan is dat de speciale symbolen binnen zo'n sublijst hun bijzondere betekenis verliezen.) Voorbeelden:

```
| ?-match([[piet,is],de,[broer,van,jan]],[piet,is,de,broer,van,jan]).
no
| ?-match([[piet,is],de,[broer,van,jan]],[[piet,is],de,[broer,van,jan]]).
yes
| ?-match([[piet,?],de,[broer,van,jan]],[[piet,is],de,[broer,van,jan]]).
no
| ?-match([[piet,?],de,[broer,van,jan]],[[piet,?],de,[broer,van,jan]]).
yes
| ?-match([[*],de,[+,jan]],[[piet,is],de,[broer,van,jan]]).
no
```

We voeren nog een conventie in: moet één van de speciale tekens letterlijk in ***L*** voorkomen, dan kan dat in het patroon geneutraliseerd worden door middel van de functor ***n***. Bijvoorbeeld:

```
| ?-match([s,=,a,n(+),b,n(?)],[s,=,a,+,b,?]).
yes
| ?-match([n(?)],[n(?)]).
no
```

De definitie van **match**, waarin ***P*** en ***L*** recursief doorlopen worden, is veel korter dan bovenstaande beschrijving! Ga stap voor stap te werk. Test elke tussenoplossing zorgvuldig uit alvorens aan de volgende stap te beginnen.

(a) Schrijf eerst een *recursieve* definitie voor het eenvoudigste geval.
(b) Voer daarna **?** in;
(c) vervolgens **+** en *****;
(d) en tenslotte de neutraliseringsfunctor, die uitsluitend de drie speciale tekens en zichzelf mag neutraliseren: bijvoorbeeld ***n(n(*))*** in het patroon correspondeert met ***n(*)*** in de lijst. (\= is nodig!).

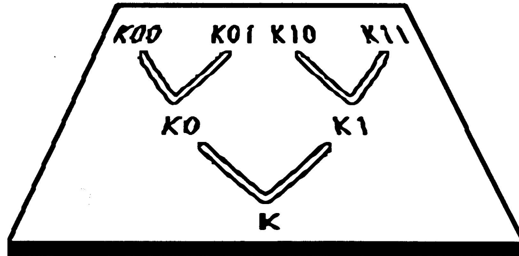
5.2 Recursiviteit en bomen

De bomen die we hier zullen bespreken zijn erg regelmatig: binair of ternair en overal even diep. Later zullen we ook andere bomen (bijvoorbeeld ontleedbomen) recursief doorlopen.

5.2.1 Een boom gerepresenteerd als losse feiten

Een predikaat is, zoals we in de inleiding van dit hoofdstuk hebben opgemerkt, ook recursief als het "zichzelf tegenkomt" via een of meer andere predikaten. (Let wel, een predikaat dat gebruik maakt van *andere* predikaten die recursief zijn, is daardoor nog niet zelf recursief.) Het predikaat weg op de volgende bladzijde lijkt op het eerste gezicht niet recursief, maar **weg** komt voor in de definities van zowel **linksaf** als **rechtsaf**.

Het predikaat **weg** berekent hoe je moet afslaan (*l*=links, *r*=rechts) om van een knoop in een *binaire boom* te komen bij een knoop die "dieper" in de boom ligt. De boom wordt gerepresenteerd met het predikaat **knoop/3**, waarbij het eerste argument een knoop van de boom aangeeft, het tweede de knoop aan het einde van de linker tak, en het derde de knoop aan het einde van de rechter tak. De **knoop**-clausules definiëren dus de volgende boom:



Iets wat we nog niet eerder zagen, is dat er twee *alternatieve reducerende stappen* zijn (de linker of de rechterhelft van de boom wordt gekozen; de linkerhelft wordt altijd het eerst uitgespit). Soms is voor de oplossing van een probleem een *conjunctie van reducerende stappen* nodig (zie opgave *5.11*). In beide gevallen spreken we van *dubbelrecursieve predikaten.*

```
weg(Van,Naar,W):-linksaf(Van,Naar,W).
weg(Van,Naar,W):-rechtsaf(Van,Naar,W).

linksaf(Van,Naar,[l]) :-knoop(Van,Naar,_).
linksaf(Van,Naar,[l|W]) :-
      knoop(Van,Links,_),
      weg(Links,Naar,W).

rechtsaf(Van,Naar,[r]) :-knoop(Van,_,Naar).
rechtsaf(Van,Naar,[r|W]) :-
      knoop(Van,_,Rechts),
      weg(Rechts,Naar,W).

knoop(k,k0,k1).
knoop(k0,k00,k01).
knoop(k1,k10,k11).
```

5.2.2 Opgave

5.9 (a) Zet de definities voor weg in de file *weg.pro* en onderzoek de werking ervan.
(b) Laat van elk van de knopen op het laagste niveau weer twee takken vertrekken, zodat er aan de boom 8 knopen (k000, k001, etc.) worden toegevoegd.
(c) De boom uit de tekst is *bi*nair, omdat vanuit elke niet-eindknoop *twee* takken vertrekken. Schrijf een programma, dat hetzelfde doet als **weg**, maar dan voor *ternaire* bomen. dat zijn bomen, waarbij van elke knoop *drie* takken vertrekken (*l*, *m* en *r*).

5.2.3 Bomen gerepresenteerd als structuur

Hieronder representeren we de binaire boom van opgave *5.9* als *structuur*. We maken gebruik

van een drieplaatsige functor ***boom***. Het eerste argument bevat de naam van de knoop, het tweede argument bevat de linker subboom, het derde argument bevat de rechter subboom. Als een subboom leeg is, staat er ***leeg*** op die plaats.

```
boom(k,
     boom(k0,
          boom(k00,
               boom(k000,leeg,leeg),
               boom(k001,leeg,leeg)),
          boom(k01,
               boom(k010,leeg,leeg),
               boom(k011,leeg,leeg))),
     boom(k1,
          boom(k10,
               boom(k100,leeg,leeg),
               boom(k101,leeg,leeg)),
          boom(k11,
               boom(k110,leeg,leeg),
               boom(k111,leeg,leeg)))).
```

De boomstructuur is recursief: elke *boom* bestaat uit een knoop, die een linker en een rechter *boom* bevat. Bij deze recursieve datastructuur schrijven we een (dubbel)recursief programmaatje **deelboom**/2, dat gegeven een knoop de bijbehorende boom retourneert. Het eigenlijke werk gebeurt door het hulppredikaat **deelboom**/3. Dit predikaat wordt door **deelboom**/2 aan het werk gezet met een hulpargument dat gelijk is aan de gehele boom. Is de ***Knoop*** van die boom gelijk aan de meegegeven knoop (stopclausule), dan wordt het hulpargument geünificeerd met het derde argument. Hebben we de boom nog niet gevonden, dan wordt de linkerhelft (***L***) onderzocht, en vervolgens de rechterhelft (**R**).We zien in dit programma weer een *hulpargument* optreden, alleen fungeert het dit keer niet als stapel, maar wordt er bij elke reducerende stap in gesnoeid.

```
deelboom(Knoop,Boom):-
     boom(K,L,R),
     deelboom(Knoop,boom(K,L,R),Boom).

deelboom(Knoop,boom(Knoop,L,R),boom(Knoop,L,R)).
deelboom(Knoop,boom(_,L,_),Boom):-
     deelboom(Knoop,L,Boom).
deelboom(Knoop,boom(_,_,R),Boom):-
     deelboom(Knoop,R,Boom).
```

5.2.4 Opgaven

5.10 Zet de definities uit de tekst in de file *deelboom.pro* en probeer ze uit. Bekijk traces en zet variabelen op plaatsen in vragen waar ze niet horen.

5.11 Schrijf in de file *deelboom.pro* een (dubbelrecursief) programma **knopen**/2, dat alle knopen van een boom in een lijstje zet.(Tip: de knopen van de linker- en die van de rechterhelft moeten worden gezocht, en samengevoegd). Een voorbeeld:

```
| ?-deelboom(k01,B),  knopen(B,Knopen).
Knopen=[k01,k010,k011]
yes
```

6 Besturingspredikaten

Een besturingspredikaat is een predikaat dat geen andere functie heeft dan het beïnvloeden van het inferentieproces.

6.1 Het cut predikaat: !/0

Het belangrijkste besturingspredikaat is het *cut* predikaat, dat voorgesteld wordt door een *uitroepteken* "!". Het *cut* -predikaat *snijdt* bij het backtracken wegen *af* en het kan daardoor het zoeken naar oplossingen versnellen. Nog belangrijker is dat met behulp van het *cut* predikaat foutieve wegen kunnen worden afgesneden. Vooral bij recursieve predikaten is dat van belang. Vanaf nu zullen we het ook vaak hebben over het *uitroepteken*, waarmee dan het *cut* predikaat bedoeld wordt. Let wel: het *uitroepteken* moet zoals elk ander predikaat door een komma of puntkomma van andere predikaten gescheiden worden. Wat doet de interpretator wanneer de doelpropositie een *uitroepteken* is?

(1) Een *uitroepteken* is de eerste keer altijd afleidbaar (**true**).

(2) *Backtrackt* de interpretator naar een uitroepteken, moet er met andere woorden een nieuwe afleiding gezocht worden voor het uitroepteken, dan veroorzaakt dat direct een **fail** (d.w.z: geen andere afleiding), niet alleen van het *uitroepteken*, ook niet alleen van de *clausule* waar het uitroepteken in staat, maar van het *predikaat*, waar de clausule met het uitroepteken deel van uitmaakt. *Andere clausules van het* ***predikaat*** *worden niet meer geprobeerd: zij vormen de afgesneden wegen* . Let wel: bij een recursief predikaat blijven de hoger in een afleiding voorkomende toepassingen van dat predikaat *onaangetast*. Het gaat alleen om de alternatieven op één niveau!

Een voorbeeld:

```
is_man(claus).
is_man(pieter).
is_vrouw(beatrix).
is_vrouw(wilhelmina).

geslacht(X,man):-is_man(X), !.
geslacht(X,vrouw):-is_vrouw(X).
```

De bedoeling van het predikaat **geslacht/2** is om van een *gegeven* persoon het geslacht op te vragen (of te controleren). Is dat geslacht mannelijk (dus slaagt **is_man(X)**), dan is het overbodig om de andere mogelijkheid te proberen. Daarom staat er in de eerste clausule van **geslacht** een uitroepteken. *Failt* **is_man(X)** in de eerste clausule, dan wordt het uitroepteken niet gezien door de interpretator, zodat de tweede clausule van **geslacht** geprobeerd wordt.

Als we tracen zien we (de *match*-regels zijn verwijderd):

```
|| ?-geslacht(claus,G).
[1] 0 Call: geslacht(claus,G)
        [2] 1 Call: is_man(claus)
        [2] 1 Exit: is_man(claus)
        [3] 1 Call: !
        [3] 1 Exit: !                      <-- uitroepteken verwerkt
[1] 0 Exit: geslacht(claus,man)
G=man;                                     <-- puntkomma ingetypt
                                           <-- Redo van uitroepteken onzichtbaar
        [2] 1 Pop: is_man(claus)           <-- Meteen terug naar
[1] 0 Pop: geslacht(claus,man)             <-- hoogste niveau
no
```

Als we het uitroepteken uit de definitie van geslacht verwijderen, krijgen we:

```
||?-geslacht(claus,G).
[1] 0 Call: geslacht(claus,G)
        [2] 1 Call: is_man(claus)
        [2] 1 Exit: is_man(claus)
[1] 0 Exit: geslacht(claus,man)
G=man;                                     <-- puntkomma ingetypt
        [2] 1 Pop: is_man(claus)
        [2] 1 Redo: is_man(claus)          <-- Redo en geen pop als boven
        [2] 1 Fail: is_man(claus)
[1] 0 Pop: geslacht(claus,man)
[1] 0 Redo: geslacht(claus,G)              <-- Redo !!
        [2] 1 Call: is_vrouw(claus)        <-- Overbodige tak van geslacht
        [2] 1 Fail: is_vrouw(claus)
[1] 0 Pop: geslacht(claus,vrouw)
no                                         <-- Eindelijk !
```

Een gevolg van het gebruik van het uitroepteken in de definitie van **geslacht** is dat dit predikaat niet geschikt is voor het vinden van alle mannen in de gegevensbank. Vandaar dat aan het begin van de bespreking van dit voorbeeld nadrukkelijk over een *gegeven* persoon gesproken werd.

```
| ?-geslacht(M,man).
M=claus;                                   <-- puntkomma ingetypt
no
```

Het uitroepteken verbiedt namelijk het zoeken van een nieuwe oplossing voor **is_man(X)**. De definitie van **geslacht** laat wél het vinden van meer vrouwen toe (probeer maar uit). Dit komt doordat de tweede clausule geen uitroepteken bevat, zodat er wel gebacktrackt wordt naar **is_vrouw(X)**. Voor de uniformiteit is het beter aan de tweede clausule van geslacht ook een uitroepteken toe te voegen:

```
geslacht(X,man):-is_man(X),  !.
geslacht(X,vrouw):-is_vrouw(X),  !.
```

Voor de volledigheid zij vermeld dat de predikaten **is_man** en **is_vrouw** wél alle mannen respectievelijk vrouwen opleveren.

Uit dit voorbeeld blijkt, dat we er bij het definiëren van een predikaat met behulp van een uitroepteken altijd op bedacht moeten zijn dat bepaalde toepassingen van dat predikaat

onmogelijk worden gemaakt, omdat er dan bepaalde eisen worden gesteld aan het al of niet variabel zijn van de argumenten. Door toepassing van het *cut* predikaat wordt het non-deterministische karakter van Prolog aangetast. Dit is er dan ook de reden van dat er naar wegen gezocht wordt het *cut* predikaat uit te bannen, zonder dat Prolog haar praktisch nut verliest.

Nu een *(dubbel)recursief* voorbeeld, het predikaat **schrap/3**: het eerste argument is een element dat verwijderd moet worden uit de lijst in het tweede argument; het resultaat komt in het derde argument:

```
| ?-schrap(x,[a,x,b,x,x,c],R).
R=[a,b,c];                        <-- puntkomma,
no                                <-- er zijn uiteraard geen andere oplossingen
```

We redeneren als volgt.
(1) Is de lijst (in het eerste element) leeg, dan is het resultaat de lege lijst.
(2) Is het eerste element van de lijst gelijk aan het te schrappen element, dan is het resultaat gelijk aan de "gereinigde" staart.
(3) In andere gevallen is het resultaat gelijk aan de lijst bestaande uit het eerste element van de gegeven lijst met daarachter de "gereinigde" staart van de gegeven lijst.

We schrijven dus op (we gebruiken om redenen die zullen blijken de naam **chrap**)

```
chrap(_,[],[]).
chrap(E,[E|T],R):-chrap(E,T,R).
chrap(E,[X|T],[X|R]):-chrap(E,T,R).
```

en we proberen of het predikaat werkt:

```
| ?-chrap(x,[a,x,b,x,c],R).
R=[a,b,c];                  <-- puntkomma
R=[a,b,x,c];                <-- ¿Een andere oplossing?
R=[a,x,b,c];                <-- ¿Nog één ??
R=[a,x,b,x,c];              <-- ¿En nog één ???
no
```

We zien tot onze schrik dat er meer dan één oplossing blijkt te zijn, waarvan alleen de eerste goed is. De oorzaak hiervan is dat een doelpropositie die met de *tweede* clausule van **chrap** unificeerbaar is, óók met de *derde* clausule unificeerbaar is, want de variabelen *X* en *E* daarin mogen best gelijk zijn. Het gevolg is dat wanneer de lijst niet leeg is *beide* clausules een oplossing leveren, waarvan de oplossing geproduceerd via de derde clausule fout is, omdat het eerste element (*X*) daar in het resultaat wordt opgenomen. De remedie is die foutieve weg af te snijden: zodra de tweede clausule toepasbaar blijkt, mag er geen andere mogelijkheid meer geprobeerd worden. Aan de tweede clausule moet daarom een *uitroepteken* worden toegevoegd.

```
schrap(_,[],[]).
schrap(E,[E|T],R):-!, schrap(E,T,R).
schrap(E,[X|T],[X|R]):-schrap(E,T,R).
```

Algemene regels voor het gebruik van het *cut* predikaat zijn moeilijk te geven: veel oefenen is het beste, vandaar de volgende opgaven.

6.1.1 Opgaven

Om een te groot automatisme te voorkomen, bevinden er zich onder de te definiëren predikaten een aantal waarbij het uitroepteken niet nodig is. Gebruik waar nodig de predikaten in *lisp.pro*.

6.1 Wat zou er veranderen aan de gebruiksmogelijkheden van **is_elem_van/2** als we aan de definitie een uitroepteken zouden toevoegen, zoals hieronder (andere naam!)?

```
is_element_van(X,[X|_]):-  !.
is_element_van(X,[_|T]):-  is_element_van(X,T).
```

6.2 Schrijf in *lisp.pro* een predikaat **na_elem/3**, dat van een lijst alle elementen na het *eerste* voorkomen van een bepaald element retourneert in een lijst. Ter verduidelijking:

```
| ?-na_elem(t,[p,l,e,t,t,e,n],Uitgang).
Uitgang=[t,e,n];                          /* Elementen na de eerste t */
no
| ?-na_elem(plaats,[naam,'Mariken',plaats,'Nieumeghen'],[N|_]).
N=Nieumeghen
yes
```

6.3 In deze opgave definiëren we in de file *verzamel.pro* enige predikaten voor verzamelingtheoretische operaties. Een verzameling bestaat uit unieke elementen. We representeren een verzameling als een lijst.

(a) Schrijf een predikaat **maak_verzameling/3**, dat ervoor zorgt dat elk element in een lijst maximaal één keer voorkomt:

```
| ?-maak_verzameling([b,c,c,b,d,c,a,a,b],V).
V=[b,c,d,a]
yes
```

In de volgende opgaven hoeft door de predikaten niet gecontroleerd te worden of de lijsten inderdaad verzamelingen zijn.

(b) Een verzameling V is een deelverzameling van een verzameling W, als elk element van V ook een element van W is. Definieer het predikaat **deelverzameling/2** dat onderzoekt of de verzameling in het eerste argument een deelverzameling is van het tweede argument:

```
| ?-deelverzameling([a,b,c],[c,d,b,a]).
yes
| ?-deelverzameling([],[c,d,b,a]).
yes
| ?-deelverzameling([a,b,p],[c,d,b,a]).
no
```

(c) In een verzameling speelt de volgorde van de elementen geen rol. Definieer **identiek/2**, dat onderzoekt of twee verzamelingen identiek zijn.

```
| ?-identiek([a,b,d,c],[b,c,d,a]).
yes
| ?-identiek([a,b,d,c],[a,b,c]).
no
```

In de voorbeelden hieronder hadden de elementen van de verzamelingen *D*, *V*, *S* en *C* ook in een andere volgorde kunnen staan.

(d) De **doorsnede** van twee verzamelingen is de verzameling van elementen die in beide verzamelingen zitten.

```
| ?-doorsnede([a,b,c,d],[a,d,e,f],D).
D=[a,d]
yes
```

(e) De **vereniging** van twee verzamelingen is de verzameling van elementen die in de eerste *of/en* de tweede verzameling zitten.

```
| ?-vereniging([a,b,c,d],[a,d,e,f],V).
V=[a,b,c,d,e,f]
yes
```

(f) Het **symmetrisch_verschil** van twee verzamelingen bestaat uit de verzameling van elementen die wel in de *vereniging* maar niet in de *doorsnede* zitten.

```
| ?-symmetrisch_verschil([a,b,c,d],[a,d,e,f],S).
S=[b,c,e,f]
yes
```

(g) Het **complement** van een verzameling ten opzichte van een andere verzameling bevat de elementen die wel in de eerste, maar niet in de tweede verzameling zitten.

```
| ?-complement([a,b,c,d],[a,d,e,f],C).
C=[b,c]
| ?-complement([a,d,e,f],[a,b,c,d],C).
C=[e,f]
```

(h) Tot slot een kleintje: een verzameling is leeg als hij geen elementen bevat.

```
| ?-leeg([]).
yes
| ?-leeg([[]]).
no
```

6.4 (a) Definieer (in *lisp.pro*) het predikaat **vervang_1/4**, dat een bepaald element in een lijst vervangt door een ander element. De vervanging moet plaatsvinden overal waar dat element voorkomt, doch alleen op het hoogste niveau (vergelijk opgave b):

```
| ?-vervang_1(el,x,[el,b,el,[el,[el],el],d],Hoogste).
Hoogste=[x,b,x,[el,[el],el],d]
yes
```

(b) Definieer vervolgens het predikaat **vervang_n**, dat een bepaald element in een lijst overal vervangt door een ander element, ook als het element in een sublijst staat.

```
| ?-vervang_n(el,x,[el,b,el,[el,[el],el],d],Alle).
Alle=[x,b,x,[x,[x],x],d]
yes
```

6.5 Definieer een predikaat **plet** (*flatten* in LISP), dat ervoor zorgt dat er in een lijst nog maar 1 niveau over is. Met andere woorden, alle haakjes binnen de lijst verdwijnen (let goed op wat er met de lege lijst gebeurt!):

```
| ?-plet([a,[b,c],[],d,[e,[f,[g]]],h],P).
P=[a,b,c,d,e,f,g,h]
yes
```

6.2 fail/0

Een systeempredikaat dat altijd "mislukt" en dat dus direct tot backtracking aanleiding geeft is **fail**, een predikaat zonder argumenten. **Fail** kan onder meer gebruikt worden om het zoeken naar meer oplossingen te forceren en om predikaten *negatief te definiëren* in termen van andere predikaten.

6.2.1 Het genereren van alle oplossingen

Een zeer belangrijke toepassing van **fail** is die, waarbij de functie van het intypen van een puntkomma na een oplossing als het ware in het predikaat wordt ingebouwd. De structuur van de condities waarin **fail** op die manier gebruikt wordt, zien we in het volgende voorbeeld:

```
toon_broers(X):-
        broer(X,Broer),
        schrijf_naar_scherm(Broer),
        fail.
toon_broers(_).
```

Een voorbeeld van het gebruik van dit predikaat is:

```
| ?-toon_broers(willem_alexander).
johan_friso
constantijn
yes
```

toon_broers werkt als volgt: het eerste conjunct zoekt een **broer** van *X*. Vooruitlopend op de stof van het volgende hoofdstuk gebruiken we het predikaat **schrijf_naar_scherm** om de gevonden ***Broer*** naar het scherm te schrijven. Daarna komen we bij **fail**. Dit veroorzaakt een *backtrackingstap,* waardoor een alternatieve afleiding voor **schrijf_naar_scherm** gezocht wordt. Deze is er niet (zoals in hoofdstuk 8 wordt uitgelegd). Daarom wordt er, na nog een stap terug, naar een volgende oplossing voor ***Broer*** in het eerste conjunct gezocht. Die wordt gevonden en ook naar het scherm geschreven. Daarna komen we weer bij **fail**, passeren we op de terugweg het schrijfpredikaat en komen we weer bij het eerste conjunct, dat een volgende broer op moet leveren. Die is er niet, waarmee de eerste clausule van **toon_broers** *failt*. We zouden het hierbij hebben kunnen laten, maar *het is belangrijk een predikaat niet onnodig te laten failen, omdat het dan niet in een conjunctie kan worden opgenomen* . Vandaar de tweede clausule die altijd slaagt (onder voorwaarde dat er precies één argument is).

6.2.2 Het negatief definiëren van predikaten

Een voorbeeld van *negatief definiëren* is:

```
klinker(Letter):-is_elem_van(Letter,[e,i,a,o,u]).

medeklinker(Letter):-klinker(Letter), !, fail.
medeklinker(_).
```

klinker onderzoekt of een meegegeven letter een klinker is. We definiëren **medeklinker/1** met behulp van **klinker**: is het door medeklinker te onderzoeken argument een klinker, dan moet *no* geantwoord worden, vandaar **fail** in de eerste regel van de definitie van **medeklinker.** Het uitroepteken is nodig om te voorkomen dat de tweede regel van de definitie geprobeerd wordt. Let erop dat het *uitroepteken niet na* **fail** kan staan, omdat bij een conjunctie de interpretator niet voorbij de **fail** kijkt! Als het aan **medeklinker** meegegeven argument géén klinker is, dan resulteert **klinker(Letter)** in de eerste regel in een *fail.* Maar dit geschiedt vóór het uitroepteken, waardoor de tweede clausule van **medeklinker** op afleidbaarheid wordt onderzocht. En dit feit heeft geen andere taak dan een positief resultaat te produceren, hetgeen gerealiseerd wordt door een anonieme variabele als argument in het feit op te nemen. Het gevolg hiervan is, dat bijvoorbeeld ook het atoom ***klinker*** als een medeklinker wordt beschouwd.

6.2.3 Opgaven

6.6 Verander de definities voor **klinker/1** en **medeklinker/1** zodanig dat *y* zowel een klinker als een medeklinker is. Zet de definities in de file *klinkers.pro.*

6.7 Definieer in de file *lisp.pro* de functie **is_geen_element_van/2**.

6.8 Definieer in de file *stamboom.pro* de relaties **broer/2**, **zus/2**, **oom/2** en **tante/2** (met een te definiëren hulppredikaat **ongelijk/2**, dat slaagt als de twee argumenten niet unificeerbaar zijn).

6.9 Het in het vorige hoofdstuk gemaakte predikaat **voor/3** zal ook slagen in gevallen als:

```
| ?-voor(x,y,[a,y,x,y,b]).
yes
```

Definieer **eerder/3** dusdanig, dat het eerste element niet alleen voor het tweede moet staan, maar ook niet na het tweede mag voorkomen.

```
| ?-eerder(x,y,[a,y,x,y,b]).
no
```

6.3 true/0

Het aan **fail** tegenovergestelde predikaat is **true** (zonder argumenten). **true** slaagt altijd direct. **true** wordt zelden toegepast, maar soms is het nuttig in disjuncties. **true** *slaagt niet meer dan één keer*, dus bij backtracking wordt er voorbij **true** gebacktrackt. Dit is anders bij het volgende predikaat.

6.4 repeat/0

Het laatste besturingspredikaat is **repeat**. Net als **true** slaagt dit predikaat direct, maar bovendien blijft het ook slagen als de interpretator door backtracking bij dit predikaat terugkeert. Dit predikaat maakt *iteratie* (herhaling) van stappen mogelijk. Een oneindige herhaling zien we hieronder:

```
oneindige_herhaling:-repeat, fail.
```

Het is de kunst om **repeat**, **fail** en *uitroepteken* zodanig te combineren dat de herhaling op de juiste wijze eindigt. Een mogelijke programmastructuur is de volgende:

```
herhaal_iets_nuttigs:-
        repeat,
        iets_nuttigs(X,Y).

iets_nuttigs(X,Y):-
        doe_iets1(X),
        doe_iets2(Y),
        !,
        fail.
iets_nuttigs(_,_).
```

iets_nuttigs wordt herhaald, zolang het **faalt**. Het echte werk gebeurt in de predikaten **doe_iets1** en **doe_iets2**. Als die beide slagen, volgen een uitroepteken en een **fail**. Het uitroepteken zorgt ervoor dat de tweede clausule van **iets_nuttigs** niet aan bod komt, zodat **iest_nuttigs** failt, wat de bedoeling was. Gaat er in **iets_nuttigs** iets mis vóór het uitroepteken, dan *slaagt* dat predikaat en wordt het iteratieve proces (of, met een Engelse term, de *loop*) gestopt.

Wat ten onrechte weleens geprobeerd wordt, is om een oplossingengenerator samen met een **repeat** te gebruiken om één *loop* te schrijven. Fout is bijvoorbeeld:

```
fout(X):-                      /* FOUT !!! */
        repeat,                                 /* Begin van buitenloop */
                broer(X,Broer),                 /* Begin van binnenloop */
                schrijf_naar_scherm(Broer),
                fail.                           /* Einde binnenloop */
```

In de definitie worden door de proposities vanaf **broer** tot en met **fail** alle broers op het scherm getoond. Als alle broers geweest zijn, resulteert **broer(X,Broer)**in een **fail**. Er wordt gebacktrackt naar de **repeat**, hetgeen veroorzaakt dat de broers nogmaals op het scherm verschijnen, enzovoorts. (In de meeste Prologimplementaties kan ^C (control C) gebruikt worden om een afleiding *af* te *breken*.) In de definitie werd vergeten, dat de interpretator voor een doelpropositie waarnaar gebacktrackt wordt, altijd alternatieve oplossingen zoekt.

7 Rekenen

7.1 Arithmetische expressies

Ofschoon Prolog in eerste instantie ontworpen is, om complexe datastructuren zoals bomen en lijsten te manipuleren, beschikken de meeste Prologimplementaties over krachtige rekenfaciliteiten. Voor rekenwerk bestaat er in Prolog een klasse van *evalueerbare functoren*. Een rekenfunctor gedraagt zich, *mits in de juiste context gebruikt*, als een *functie* die *geëvalueerd* wordt: in tegenstelling tot normale predikaten is het resultaat niet *fail* of *true*, maar een *getal*. Meestal wordt een rekenfunctor gebruikt in de vorm van *operator*, waardoor *arithmetische expressies* er uit zien zoals we gewend zijn. In het volgende overzicht staan de rekenfunctoren zowel in de operator- als in de functornotatie:

operatornotatie	*functornotatie*	*resultaat van evaluatie*
X+*Y*	+(*X*,*Y*)	Som van *X* en *Y*
X+*Y*	-(*X*,*Y*)	Verschil van *X* en *Y*
*X***Y*	*(*X*,*Y*)	Produkt van *X* en *Y*
X/*Y*	/(*X*,*Y*)	Quotient van *X* en *Y* (met decimalen!)
X//*Y*	//(*X*,*Y*)	Quotient van *X* en *Y* (zonder decimalen!)
X mod *Y*	mod(*X*,*Y*)	*X* modulo *Y* (Rest van *X* na deling door *Y*)

X en *Y* in de expressies hierboven staan voor een *getal*, een *expressie* (met andere woorden complexe expressies zijn mogelijk) of een *variabele*. Een variabele moet op het moment van evaluatie gebonden zijn aan een evalueerbare expressie. Bij de evaluatie van een expressie in functornotatie worden eerst de argumenten geëvalueerd, vervolgens wordt op de resultaten daarvan de rekenoperatie toegepast. Bij de evaluatie van een expressie in operatornotatie worden de gebruikelijke prioriteiten aangehouden: eerst worden de *subexpressies tussen haakjes* en de *expressies in functornotatie* geëvalueerd, daarna volgt toepassing van de **mod** operator, dan vermenigvuldigen (*) en delen (// en /), tenslotte optellen (+) en aftrekken (-). **mod** is *niet associatief*, hetgeen wil zeggen dat **mod** in een expressie niet zonder het gebruik van haakjes twee keer achter elkaar als operator in een expressie voor mag komen. De overige operatoren zijn *links-associatief*: dit impliceert dat de volgorde van toepassing van operatoren met gelijke prioriteit van links naar rechts verloopt. *Haakjes* mogen worden gebruikt om de normale volgorde van evaluatie te wijzigen. Uit de voorbeelden hieronder blijkt dat de operatornotatie in de meeste gevallen de duidelijkste is.

operatornotatie	*functornotatie*	*resultaat van evaluatie (en commentaar)*	
10//3	**//(10,3)**	**3**	(geen cijfers achter komma!)
3//10	**//(3,10)**	**0**	(idem)
10 mod 3	**mod(10,3)**	**1**	(want 10=3*3 *rest* 1)
3 mod 10	**mod(3,10)**	**3**	(want 3=0*10 rest 3)

2*4-8	**-(*(2,4),8)**	**0**	
2*(4-8)	***(2,-(4,8))**	**-8**	
16/4/2	**/(/(16,4),2)**	**2**	(van links naar rechts)
16/(4/2)	**/(16,/(4,2))**	**8**	
2*5 mod 4	**mod(*(2,5),4)**	**2**	(*beide notaties*: mod(2*5,4))
6 mod (5 mod 3)	**mod(6,mod(5,3))**	**0**	(eerst wat tussen haakjes staat)
(6 mod 5) mod 3	**mod(mod(6,5),3)**	**1**	(eerst wat tussen haakjes staat)

7.2 Context van evalueerbare expressies

Evaluatie van een arithmetische expressie vindt alleen plaats in de context van een *numerieke vergelijking* of in de context van het is predikaat. Dit verklaart de volgende sessie:

```
| ?-2*4-8.
no.
| ?-mod(*(2,5),3).
no
| ?-3+2=2+3.
no.
```

In de twee eerste voorbeelden worden **-(*(2,4),8)** en **mod(*(2,5),3)** niet geëvalueerd, maar op *afleidbaarheid* onderzocht. In het laatste voorbeeld wordt getest of **+(3,2)** *unificeerbaar is met* +(2,3) en dat is niet het geval. (Om numerieke gelijkheid van expressies te onderzoeken moet =:= gebruikt worden, zie de volgende paragraaf).

7.2.1 Numerieke vergelijkingen

Numerieke vergelijkingen ontstaan met behulp van vergelijkingsoperatoren (*E1* en *E2* zijn arithmetische expressies):

operatornotatie	*functornotatie:*	*true indien geldt:*
E1* =:= *E2	**=:=(*E1*,*E2*)**	*E1* is gelijk aan *E2*
E1* =\= *E2	**=\=(*E1*,*E2*)**	*E1* is ongelijk aan *E2*
E1* > *E2	**>(*E1*,*E2*)**	*E1* is groter dan *E2*
E1* < *E2	**<(*E1*,*E2*)**	*E1* is kleiner dan *E2*
E1* >= *E2	**>=(*E1*,*E2*)**	*E1* is groter dan of gelijk aan *E2*
E1* =< *E2	**=<(*E1*,*E2*)**	*E1* is gelijk aan of kleiner dan *E2*

Een numerieke vergelijking is maximaal één keer succesvol. Bij een backtrackingstap zal een vergelijking dus altijd tot nog een *stap terug* aanleiding geven. Een voorbeeldsessie:

```
| ?-3+2 =:= 2+3.
yes                              /* Hoera ! */
| ?-10 mod 2 =:= 0.
yes
| ?-/(/(16,4),2)) =\= 16/4/2.
no
```

```
| ?-[user].
positief(A,B,C) :- B*B-4*A*C > 0.
^Z
yes
| ?-positief(2,5,2).
yes
```

Een numerieke vergelijking is alleen mogelijk als beide argumenten *op het moment van vergelijking* een numerieke waarde hebben, hetgeen in het volgende voorbeeld niet het geval is:

```
| ?-B*B-4*A*C>0.
Illegal arithmetic expression
```

7.2.2 is/2

Om een variabele te binden aan de waarde van een numerieke expressie moet gebruik gemaakt worden van het **is/2** predikaat, hetgeen veelal gebeurt in de operatornotatie:

***Variabele* is *Expressie*,**

hetgeen equivalent is met het veel minder gebruikte **is(*Variabele,Expressie*)**. Net als de numerieke vergelijkingen is ook **is** maximaal één keer succesvol.

```
| ?-C1 is 39//10.
C1=3
yes
| ?-C2 is 39 mod 10.
C2=9
yes
| ?-A=3, F is A+A, E is F*F, W is E/2.
A=3,
F=6,
E=36,
W=18
yes
```

Een expressie mag variabelen bevatten mits deze geünificeerd zijn met evalueerbare termen:

```
| ?-F=A+A, E=F*F, A=3, W is E/2.
F=3+3,
E=(3+3)*(3+3),
A=3,
W=18
yes
```

Bij het werken met getallen kunnen conjuncten vaak niet in willekeurige volgorde staan:

```
| ?-F=A+A, E=F*F, W is E/2, A=3.
Illegal arithmetic expression          /* Namelijk is(W,(A+A)*(A+A)/2) */
```

Tot slot volgt bij wijze van voorbeeld de definitie van een recursief predikaat dat de cijfers van een getal ≥ 0 (!), in een lijst zet. Een voorbeeld:

```
| ?-getal_lijst(32567,L).
L=[3,2,5,6,7]
yes
```

We zien in de definitie hieronder een verschijnsel dat bij recursieve predikaten vaker voorkomt: de elementen van de gezochte lijst worden in omgekeerde volgorde gevonden, waardoor de lijst na het echte werk nog even omgekeerd moet worden.

```
getal_lijst(G,L):-
        getal_lijst_r(G,R),      /* Geeft lijst in omgekeerde volgorde */
        reverse(R,L).            /* Omkeren dus */

getal_lijst_r(G,[G]):-           /* Nog één cijfer over */
        G =< 9,!.                /* Stopconditie */
getal_lijst_r(G,[C|T]):-
        C is mod(G,10),          /* Geeft het laatste cijfer */
        W is G//10,              /* Verwijdert het laatste cijfer */
        getal_lijst_r(W,T).      /* Zoek rest van cijfers */
```

7.3 Opgaven

7.1 Definieer een predikaat in de file *lisp.pro* **maximum/2**, dat het grootste getal in een lijstje van getallen zoekt. Idem dito voor **minimum/2**. Tip: Gebruik een tijdelijk maximum/minimum.

```
| ?-maximum([10,32,5,17],M).
M=32
yes
| ?-minimum([10,32,5,17],M).
M=5
yes
```

7.2 Definieer in de file *reken.pro* een predikaat **exp/3**, dat een getal tot een macht (≥0) verheft. N.B.:**exp** leidt gauw tot grote getallen. Elke Prologimplementatie heeft grenzen waarboven en waaronder getallen alleen nog maar bij benadering berekent kunnen worden (met behulp van *floating point* getallen). Vaak leiden te grote getallen tot negatieve getallen en omgekeerd.

```
| ?-exp(2,10,G).
1024
yes
```

7.3 Definieer in de file *lisp.pro* een predikaat **nth/3** dat het *n*-de element van een lijst retourneert. Definieer ook **nthtail/3** dat alle elementen na het *n*-de element retourneert.

```
| ?-nth([aap,noot,mies,truus],3,E).
E=mies
yes
| ?-nthtail([aap,noot,mies,truus],2,E).
E=[mies, truus]
yes
```

7.4 (a) Zet de definitie van **getal_lijst** in de file *reken.pro*. Is de variabele *W* in de tweede clausule van **getal_lijst_r** van de tekst eigenlijk wel nodig? Onderzoek wat er gebeurt als het conjunct **W is G//10** wordt weggelaten en *G//10* in het laatste conjunct als argument wordt gebruikt. Gebruik de tracer.
(b) Verander het predikaat **getal_lijst** uit de tekst zodanig, dat het ook voor negatieve getallen gebruikt kan worden.

```
| ?-getal_lijst(-12345,L).
L=[-,1,2,3,4,5]
yes
```

7.5 En nu een klein project: een *symbolische calculator*. Op personal computers is, zoals al in opgave 7.2 werd opgemerkt, het werken met getallen beperkt tot vrij kleine getallen. Om deze grens te doorbreken willen we rekenen met getallen die gerepresenteerd zijn als lijst. Wat er moet gebeuren is de papier en potlood methode die we op de lagere school geleerd hebben vertalen in een Prologprogramma. We beperken ons hierbij tot het optellen en vermenig - vuldigen van gehele getallen groter dan of gelijk aan nul ([0]).

```
| ?-gpg([9,9,9,9,9,8,9],[1,1],S).
S=[1,0,0,0,0,0,0,0]
yes
| ?-gxg([5,1,2],[2,0,4,8],P).
P=[1,0,4,8,5,7,6]
yes
```

Een belangrijk aspect bij het optellen is het onthouden van cijfers die naar de volgende kolom moeten, bijvoorbeeld: *9+8=7 (1 onthouden)*. Als bijdrage aan de *symbolische calculator* volgt hier een predikaat **cpc/5** (*cijfer plus cijfer*), dat rekeninghoudend met een *overloop uit een vorige kolom*, een nieuw *cijfer* en een *overloop* produceert:

```
cpc(I,J,O_in,C,O_out):-
        P is I+J+O_in,
        O_out is P//10,
        C is P mod 10.
```

Voorbeelden:

```
| ?-cpc(9,8,0,C,O).
C=7,
O=1
yes
| ?-cpc(9,8,1,C,O).
C=8,
O=1
yes
```

Aanwijzingen: Gebruik **cpc** om **gpg** (*getal plus getal*) te definiëren (in *calcula.pro*). Schrijf een predikaat **cxc** (*cijfer maal cijfer*); gebruik **cxc** en **gpg** om **cxg** (*cijfer maal getal*) te definiëren; definieer met behulp van **cxg** en uiteraard weer **gpg** uiteindelijk **gxg** (*getal maal getal*). Het kán handig zijn zo hier en daar vóór het echte werk begint reverse toe te passen.

8 Invoer en uitvoer

Zoals elke programmeertaal biedt Prolog faciliteiten om een programma te laten communiceren met de buitenwereld. Deze communicatie vindt plaats via de *randapparatuur*, waartoe ondermeer behoren het toetsenbord, het beeldscherm en de schijfeenheid. Er zijn systeempredikaten om een programma iets te laten vertellen (*uitvoerpredikaten*) en systeempredikaten om gegevens in te lezen (*invoerpredikaten*). In- en uitvoer samen worden meestal aangeduid met de, op zijn Engels uitgesproken, afkorting *I/Θ* ("Ai O", *Input/Output*).

De I/O predikaten vormen een bijzondere groep van predikaten: ze zijn *imperatief*. Dat wil zeggen dat ze het karakter van een bevel hebben. Het gaat bij deze predikaten niet om het normale vinden van een afleiding, maar om het *zijeffect*, wat bij deze predikaten bestaat uit het transport van gegevens. Alle I/O-predikaten hebben gemeenschappelijk dat ze nooit meer dan één keer slagen, dus bij backtracking lezen of schrijven ze niet nog een keer. Ook is het niet het geval dat bij het backtracken een lees of schrijfopdracht fysiek weer ongedaan wordt gemaakt: eenmaal gelezen of geschreven blijft gelezen of geschreven.

8.1 Term I/O-predikaten

De natuurlijke eenheid van verwerking bij Prolog is de term (zie voor de definitie van term 4.5). Voor uitvoer van een term zijn het **write/1** en het **display/1** predikaat beschikbaar, invoer van een term gaat met het **read/1** predikaat. Al deze predikaten gebruiken gewoonlijk *user* als randapparaat. Dat wil zeggen dat uitvoer naar het scherm geschreven wordt en invoer ingelezen wordt vanaf het toetsenbord. Technischer uitgedrukt: gewoonlijk fungeert het scherm als *outputstream* en het toetsenbord als *inputstream*.

8.1.1 write/1

write(*Term*) schrijft een term naar de outputstream.

Hieronder volgen voorbeelden.

```
| ?-write(23348).
23348
yes
| ?-write(socrates).
socrates
yes
| ?-write(mens(socrates)).
mens(socrates)
yes
| ?-write((5+3)*2).
(5+3)*2                                   /* Er wordt niets uitgerekend! */
yes
```

```
| ?-write(*(+(5,3),2)).
(5+3)*2                                /* Write gebruikt de operatornotatie */
yes
| ?-write(A320).
A320                                   /* Dit is de output van write */
A320=A320                              /* Dit is de "Oplossing" */
yes
| ?-X is (5+3)*2, write(X).
16
X=16
yes
| ?- write(X),
   X=term,                             /* 1e write: X heeft nog geen waarde*/
   write('->'),                        /* Schrijft een pijltje */
   write(X).                           /* 2e write: X=term*/
X->term                                /* Resultaat v.d. 3 writes */
X=term                                 /* Oplossing! */
yes
| ?-write(Fout(predikaat)).
Syntax error                           /* Predikaat begint met een hoofdletter */
| ?-write('Fout(predikaat)').
Fout(predikaat)                        /* Atoom (zie 2.8) */
yes
| ?-write([]).
[]
yes
| ?-write([a,b|[c,d]]).
[a,b,c,d]                              /* Zelfde lijst als meegegeven in write! *
yes
| ?-write('Hoofdletters, leestekens, spaties, "returns",
|                                                       /* 2 keer RETURN */
| allemaal tussen apostrophen (")!').
Hoofdletters, leestekens, spaties, "returns",
                                                        /* BLANCO REGEL */
allemaal tussen apostrophen (')!
yes
write("abcde...xyz").                  /* Aanhalingstekens -> string */
[97,98,99,100,101,46,46,46,120,121,122] /* Zie sectie 8.6 over strings */
yes
| ?-write('Syntax  error').
Syntax error                           /* Niets aan de hand! */
yes
```

8.1.2 writeq/1 (*write quoted*)

writeq(*Term*) schrijft een term naar de outputstream met *waar nodig* apostrophen (*quotes*) rondom atomen, zodat deze later met behulp van **read** of **(re)consult** gelezen kunnen worden.

```
| ?-writeq('->').
'->'
yes
```

```
| ?-writeq(' "Airbus A340" ').
' "Airbus A340" '
yes
| ?-writeq(write('Inava')),write('.').
write('Inava').
yes
| ?-writeq('parallellepipidum').
parallellepipidum
yes
```

writeq is handig voor het creëren van files die later weer ingelezen moeten worden door **read** of (re)**consult**.

8.1.3 display/1

display(*Term*) schrijft een term in functornotatie naar de outputstream.

```
| ?-display((5+3)*2).
*(+(3,5),1).
yes
| ?-display(A mod 10).
mod(A,10)
A=A
no
| ?-A=23, display(A mod 10).
mod(23,10)
A=23;
no
| ?-display('''').
''''                    /* omringende apostrophen ook naar outputstream */
yes
```

8.1.4 read/1

read(*Term*) leest een term vanaf de inputstream. De ingelezen term moet afgesloten zijn door een *punt* en *white space*. Het predikaat slaagt alleen als er een syntactisch correcte term kan worden ingelezen die unificeerbaar is met het argument van de **read** opdracht.

Als de interpretator vanaf het scherm moet lezen, verschijnt er (bij de meeste implementaties) een PROMPT ('|: ') en wacht de interpretator totdat er een term, een punt en een return zijn ingetypt (*white space* wordt genegeerd). Hieronder illustreren we **read** met verschillende typen van argument: een variabele, een term zonder variabelen en een structuur met variabelen.

8.1.4.1 Een variabele als argument

Als het argument een variabele is, wordt er een term ingelezen en de ingelezen term wordt geünificeerd met de variabele. Dit is het meest gebruikelijke geval.
Voorbeelden:

```
| ?-read(Term).
|: mens(socrates).             /*'|: ' is de prompt */
Term=mens(socrates)
yes
```

```
| ?-read(Expressie).
|: 3 + 5.                               /* Spaties op de juiste plaats mogen. */
Expressie=3+5                          /* N.B.: Er wordt niets uitgerekend! */
yes
| ?-read(Expressie),  write(Expressie).
|: 3*5+1                               /* '.' vergeten */
|: .                                   /* alsnog ingetypt */
3*5+1                                  /* output van WRITE */
Expressie=3*5+1                        /* Substitutie */
yes
| ?-read(Expressie),  display(Expressie).
|: 3 * 5 + 1.
+(*(3,5),1)                            /* output van DISPLAY */
Expressie=3*5+1
yes
| ?-read(Lijst).
|: [kalkar,handford,tsjernobyl].
Lijst=[kalkar,handford,tsjernobyl]
yes
```

8.1.4.2 Een term zonder variabelen als argument

Als het argument een term zonder variabelen is, zal **read** alleen slagen als die specifieke term ingelezen wordt, of een term waarin variabelen zitten waardoor unificatie mogelijk is. Het laatste geval komt waarschijnlijk alleen voor bij een programmeerfout. Het eerste geval kan wel handig zijn, zoals blijkt uit de definitie van het predikaat **ja_of_nee** verderop:

```
| ?-read(lijst).                       /* Argument is atoom! */
|: [kalkar,handford,tsjernobyl].
no
| ?-read(lijst).
|: lijst.
yes
| ?-read(mens(jan)).
|: mens(X).
yes
```

In het volgende voorbeeld wordt een predikaat gedefinieerd, waarmee een *ja/nee vraag* kan worden gesteld. Elk antwoord anders dan *ja* wordt uitgelegd als ***nee***. Een nadeel van dit predikaat is dat bij het beantwoorden van de vraag de *punt* niet vergeten mag worden.

```
| ?-[user].                            /* we typen eerst de definitie in: */
| ja_of_nee(Vraag) :-
|      write(Vraag),                   /* de vraag */
|      write('  [ja.|nee.]  '),        /* Aangeven van antwoordmogelijkheden */
|      read(ja).                       /* read: Is antwoord 'ja'? */
| ^Z                                   /* Einde van de input voor consult */
| ?-ja_of_nee('Is het mooi weer?').
Is het mooi weer? [ja.|nee.] |: ja.
yes
| ?-ja_of_nee('Sure?').
Sure? [ja.|nee.] |: n.                 /* n is geen no! */
no
```

8.1.4.3 Een structuur met variabelen als argument

Door aan **read** een structuur met variabelen als argument mee te geven wordt, van de in te lezen term een bepaalde vorm (*format*) verwacht.

```
| ?-read(kerncentrale(K)).        /* Het argument geeft aan dat een */
|: kerncentrale(tsjernobyl).      /* kerncentrale moet worden ingelezen */
K=tsjernobyl
yes
| ?-read(kerncentrale(K)).
|: grafietreactor(tsjernobyl)     /* Ander predikaat dan in argument */
no
| ?-read([X,Y,Z])                 /* Het argument geeft aan dat een lijst */
                                  /* met 3 argumenten verwacht wordt */
|: [
|: kalkar,                        /* Spaties en returns deren niet */
|: handford,                      /* Returns veroorzaken wel extra prompts */
|: tsjernobyl
|: ].
X=kalkar,
Y=handford,
Z =tsjernobyl
yes
```

De interpretator gebruikt **read** en **write** voor de communicatie met de gebruiker. Ook **consult** en **reconsult** gebruiken **read** en **write.** Hierdoor komt het dat we bij het invoeren van een term voor een **read** opdracht dezelfde vrijheid hebben in het gebruik van *white space* als bij het stellen van vragen en bij het definiëren van predikaten. In het volgende voorbeeld zien we dat de interpretator rapporteert met behulp van **write** en niet met behulp van **display**.

```
| ?-read(Expressie).
|: +(*(3,5),1).                   /* Functornotatie ingetypt */
Expressie=3*5+1                   /* De interpretator gebruikt zelf write!*/
yes
```

Een clausule is ook een term, met als hoogste operator het indien-symbool. In het volgende voorbeeld zien we een **read** opdracht een clausule inlezen (net zo als dat bij een **(re)consult** geschiedt). We gebruiken **display** om de clausule in de functornotatie op het scherm te tonen. De interpretator gebruikt zelf **write** om te laten zien aan welke waarde het argument van de **read** opdracht gebonden wordt:

```
| ?-read(C),display(C).
|: zoon(K,O):- kind(K,O), is_man(K).                /* Een CLAUSULE! */
:-(zoon(K_1,O_1),(kind(K_1,O_1),is_man(K_1)))      /* :- als functor! */
C=zoon(K_1,O_1):-kind(K_1,O_1),is_man(K_1)         /* :- als operator. */
yes
```

8.1.5 Een bijzonder atoom: *End_of_file*

Indien het einde van een file bereikt is en er volgt een **read** opdracht, dan slaagt de **read** alleen wanneer het argument unificeerbaar is met het atoom **end_of_file**. Het einde van een file kan

explicief aangegeven worden door een **^Z** (zoals we herhaaldelijk gezien hebben) of door het atoom ***end_of_file***, Bijvoorbeeld:

```
| ?-read(X).
|:^Z                    /* Control-Z ingetypt=einde van file */
X=end_of_file
yes
| ?-read(X).
|:end_of_file           /* end_of_file is ook duidelijk! */
X=end_of_file
yes
| ?-read(constante).
|:^Z
no                      /* no: want ingelezen term is end_of_file */
```

8.2 Opmaak: nl/0, tab/1

Voor het verzorgen van de opmaak van teksten bestaan er twee speciale systeempredikaten, **nl/0** en **tab/1**.

nl/0 (*New Line*) stuurt een *return* naar de output, waardoor alle volgende output op de volgende regel zal beginnen.

tab(*N*) (*Tabuleer*) stuurt *N* spaties naar de output.

Hieronder volgt een voorbeeld van een vraag-doe-antwoord cyclus (**sexen/0**), gerealiseerd met behulp van **repeat** en **fail**, waarin enige van de behandelde I/O predikaten toegepast worden. De bedoeling is dat er aan degene die achter het toetsenbord zit een naam gevraagd wordt en dat het systeem uitzoekt wat het geslacht is dat bij die naam behoort. De cyclus eindigt bij een **end_of_file**.

```
sexen :-
    write('Bij een voornaam wordt het geslacht gezocht'),nl,
    repeat,                         /* Begin van cyclus */
    nl,                             /* begin nieuwe regel.*/
    write('NAAM  '),
    read(Naam),
    ( Naam=end_of_file,!            /* end_of_file: stoppen */
      ;                             /* anders :*/
      geslacht(Naam,G),             /* Bepaal geslacht G */
      tab(20),                      /* Output 20 spaties*/
      write(G),                     /* Output G*/
      fail                          /* Forceer herhaling */
    ).

geslacht(X,man) :- is_man(X).
geslacht(X,vrouw) :- is_vrouw(X).
is_man(danraj).
is_man(desi).
is_man(dinesh).
is_man(vijay).
is_man(winnoot).
```

```
is_vrouw(aruna).
is_vrouw(desi).
is_vrouw(roline).
is_vrouw(samia).
is_vrouw(urmila).
```

Een voorbeeld van een mogelijke dialoog is:

```
| ?-sexen.
Bij een voornaam wordt het geslacht gezocht

NAAM |: roline.
                vrouw
NAAM |: vijay.
                man
NAAM |: fieny.                                  /* Staat niet in de gegevens */
                                                /* dus geen tab(20) en write(G) */
NAAM |: desi.
                man              vrouw          /* BEIDE ! */
NAAM |:^Z                                       /* end_of_file */
yes
```

8.2.1 Opgaven

8.1 Schrijf een predikaat **hallo/2**, dat regels produceert met daarop *'Hallo!'* (inclusief de apostrophen). Het eerste argument geeft aan hoeveel regels er uitgevoerd moeten worden, het tweede argument geeft aan *op* welke positie in de regel de kreet *'Hallo!'* moet beginnen. Dus:

```
?-hallo(4,6).
     'Hallo!'
     'Hallo!'
     'Hallo!'
     'Hallo!'
yes.
```

8.2 Schrijf een predikaat dat alle *X*'en op het scherm laat zien waarvoor geldt: **is_man(X)**. Voeg de definitie toe aan de file *stamboom.pro.*

8.3 Verander het in de tekst gedefinieerde predikaat **ja_of_nee** zodanig, dat het in te typen antwoord echt alleen maar 'ja.' of 'nee.' mag zijn. Indien het geen van beide is, moet de vraag herhaald worden, tot één van beide antwoorden is ingetypt. Als het antwoord 'ja.' is, dan moet het predikaat slagen; in het andere geval moet het fa(i)len. Zet de definitie in *vragers.pro.*

8.3 ASCII I/O-predikaten

Naast de I/O predikaten die hele termen inlezen, zijn er in Prolog ook predikaten die opereren op één enkel teken, zoals een letter, cijfer of leesteken, maar ook op onzichtbare tekens die een bijzondere functie hebben (zoals bijvoorbeeld *return* en *linefeed*). De verzameling tekens waarop de predikaten werken, bestaat uit de tekens van de *ASCII character set*. Dit is een verzameling van 256 tekens, genummerd van 0 tot en met 255. Het nummer van een teken heet de *ASCII-code* van dat teken. De ASCII-code van de hoofdletter 'A' bijvoorbeeld is 65, die van de kleine letter 'a' is 32 *hoger*, namelijk 97. Van 'B' is de code 66 en van 'b' 98. Op alle

alle computers die gebruik maken van de ASCII *character set* hoort bij elk teken *hetzelfde* nummer. Dit is van groot belang voor de uitwisselbaarheid van gegevens tussen computers van verschillende fabrikanten. ASCII staat dan ook voor: *American Standard Code for Information Interchange.* Deze achtergrond informatie is hier relevant, omdat het argument van de nu te bespreken I/O predikaten unificeerbaar moet zijn met een ASCII-code.

8.3.1 get0/1

get0(C) leest het eerstvolgende teken van de inputstream.

Wat betreft het argument ***C*** zijn er twee verschillende gebruiksmogelijkheden:

(1) Als ***C*** een variabele is, dan wordt ***C*** met de ASCII-code van het ingelezen teken geünificeerd.

(2) Als ***C*** een ASCII-code is, dan wordt gecontroleerd of het ingelezen teken dezelfde code heeft. Is dat het geval dan slaagt **get**, anders is het resultaat **fail**.

```
| ?-get0(C).
|: A
C=65
yes
| ?-get0(C).
|: a
C=97
yes
| ?-get0(65).
|: A
yes
| ?-get0(97).
|: A
no
| ?-get0(Spatie).
|:                                  /* spatie ingetypt */
Spatie=32
yes
| ?-get0(Eof).
|:^Z                                /* ^Z ingetypt=End_of_file */
Eof=26
yes
```

Een laatste opmerking: de toetscombinaties ***Control-letter*** (*control characters*) hebben soms een speciale betekenis voor de interpretator en worden daardoor *weggevangen*, zodat die niet goed ingelezen kunnen worden. (De ASCII codes voor Control-letter combinaties is gelijk aan de plaats van de letter in het alfabet, bijvoorbeeld: ^C=3, ^G=7, ^Z=26.)

8.3.2 get/1

get(*C*) leest het eerstvolgende *zichtbare teken* in de input.

Een teken is *zichtbaar* wanneer de ASCII-code groter is dan 32. Tekens met een lagere code worden door get overgeslagen. Doordat de ASCII-codes van *return*, spatie en tabulatie alle

kleiner dan 33 zijn, worden deze tekens door **get** overgeslagen en hebben we enige *whitespace* vrijheid bij het intypen van het "echte" teken.

Wat betreft het argument zijn er evenals bij **get0** twee gebruiksmogelijkheden:

(1) Als ***C*** een variabele is, dan wordt ***C*** geünificeerd met de ASCII-code van het eerste teken waarvan de ASCII-code groter dan 32 is. *Control characters* kunnen door de interpretator weggevangen worden. Inlezen van **^Z** (***end_of_file***) of het daadwerkelijk bereiken van het einde van een file heeft tot gevolg dat ***C*** de waarde 26 krijgt (de ASCII-code van ^Z).

(2) Als ***C*** een ASCII-code is (groter dan 32!), controleer dan of het ingelezen teken dezelfde code heeft. Is dat het geval dan slaagt **get**, is dat niet het geval, dan is het resultaat **fail**.

Voorbeelden:

```
| ?-get(C).
|:                    /* return ingetypt */
|:    A               /* Een aantal spaties ingetypt en dan pas A */
C=65
yes
| ?-get(Eof).
|:                    /* ^Z lager dan 32, speciaal teken! */
Eof=26
yes
```

8.3.3 skip/1

skip(*C*) leest tekens in van de inputstream totdat er een teken met een ASCII-code gelijk aan ***C*** ingelezen is (***C*** moet dus gebonden zijn). Een betere naam voor dit predikaat zou zijn *skip_until*.
Met dit predikaat kan handig naar het einde van een inputregel gesprongen worden:

```
?-skip(31).
====> Van alles en nog wat, dan een return:
yes
```

8.3.4 put/1

put(*C*) schrijft het teken met de opgegeven ASCII-code naar de outputstream (*C* moet gebonden zijn).

```
| ?-Nul=48,put(Nul),put(Nul),put(55).
007
yes
| ?-put(7).           /* ^G: bel, probeer maar eens*/
yes
```

Het volgende voorbeeld laat zien dat het mogelijk is met ASCII-codes te rekenen:

```
| ?-get(Kleineletter),
   Hoofdletter is Kleineletter-32,
   put(Hoofdletter).
|: a
A
Kleineletter=97,
Hoofdletter=65
yes
```

Het volgende voorbeeld laat het predikaat **j_n_vraag/2** zien dat een vraag op het scherm laat zien en vervolgens wacht op een antwoord, waarvan het eerste teken gelijk moet zijn aan een ***j*** of een ***n*** (*white space is toegestaan*). Als het tweede argument een variabele is, komt het gegeven antwoord daarin terecht als ***ja*** of ***nee***. Is bij toepassing van het predikaat het antwoord al ingevuld, dan slaagt het predikaat alleen als het ingetypte antwoord overeenstemt met het opgegeven antwoord.

```
j_n_vraag(Prompt,Antwoord) :-
        repeat,
        write(Prompt), write('? (j/n) '),
        get(Ingetypt),
        nl,
        check_j_n_antw(Ingetypt,Vertaling),!,
        Antwoord=Vertaling.        /* Antwoord kan Variabele zijn of gebonden!*/

check_j_n_antw(106,ja).            /* 'j' antwoord wordt vertaald in 'ja' */
check_j_n_antw(110,nee).           /* 'n' antwoord wordt vertaald in 'nee' */
```

8.3.5 Opgaven

8.4 Definieer in *letters.pro* het predikaat **is_letter/1**, dat test of een in het argument opgegeven getal de ASCII-code van een (hoofd)letter is.

8.5 Definieer (in *letters.pro*) een predikaat **ascii_codes** met 2 argumenten. Beide argumenten moeten een ASCII-code zijn. Als de eerste code kleiner is dan de tweede moeten in oplopende volgorde de ASCII-codes met de daarbij behorende tekens op het scherm verschijnen, waarbij elk code-teken-paar op een aparte regel verschijnt. (Mooiere lay-out mag natuurlijk ook.) Als de eerste code niet kleiner is dan de tweede moeten de ascii codes in afnemende volgorde op het scherm verschijnen. Bijvoorbeeld:

```
?-ascii(88,90).
88 X
89 Y
90 Z
yes
?-ascii(122,120).
122 z
121 y
120 x
yes
```

ASCII-codes onder de 33 zijn interessant, en ook die boven de 200. Ach, eigenlijk hebben ze allemaal wel wat, probeer maar eens uit.

8.6 (a) Definieer in *letters.pro* een predikaat **schuif/2**, dat een als argument meegegeven letter vertaalt in een letter die 2 plaatsen verder staat in het alfabet en deze vertaling unificeert met het tweede argument. De volgende restricties zijn van toepassing.
* De verschuiving is *cyclisch*: mocht bij het schuiven het einde van het alfabet gepasseerd worden, dan wordt weer bij de letter 'a' verder geteld.
* Andere tekens dan letters moeten onveranderd blijven.
* Hoofdletters moeten veranderd worden in (verschoven) kleine letters. Voorbeelden van omzettingen zijn (-> staat voor *wordt omgezet in*):

```
a -> c    A -> c    = -> =
b -> d    B -> d    1 -> 1
x -> z    X -> z    ? -> ?
y -> a    Y -> a
z -> b    Z -> b
```

(b) Veralgemeniseer **schuif** door er een parameter (hier verder aangeduid als *N*) aan toe te voegen die aangeeft hoeveel plaatsen moet worden opgeschoven. (*N >=0*). *N* mag zo groot zijn als de machine toelaat. Merk op dat N=2 (0x26+2), N=28 (1x26+2), N=54 (2x26+2) enzovoorts alle dezelfde vertalingen opleveren. Dit wijst erop dat in de definitie van **schuif/3** op de een of andere manier "**modulo** 26" moet worden gerekend.
(c) Maak het mogelijk een negatieve waarde aan *N* mee te geven en interpreteer dit als terugschuiven (C->a, z->x bijvoorbeeld).

8.7 Voeg het predikaat **j_n_vraag/2** toe aan de file *vragers.pro*. Breng veranderingen aan waardoor ook de antwoorden 'J', 'N', 'y' en 'Y' (van "yes") toegestaan zijn. Als er een niet toegestaan teken wordt ingetypt, moet de bel rinkelen (**put**(7)).

8.4 Stringpredikaten

Vaak willen we ingelezen gegevens analyseren op letterniveau. Van een ingetypt woord bijvoorbeeld willen we in staat zijn elke letter apart te bekijken. Dat kan als we de ASCII-codes van de letters in een lijstje zetten, dus bijvoorbeeld in plaats van met het atoom *aap* werken we met de lijst *[97,97,112]*. Er is een systeempredikaat dat de omzetting van een constante naar een lijst van ASCII-codes en omgekeerd bewerkstelligt en dat is **name/2**.

8.4.1 name/2

name(Atoom,String) zet de tekens waaruit het ***Atoom*** bestaat om in ASCII-codes en unificeert deze met de ***String***, of omgekeerd, afhankelijk van welk argument variabel is.**name** geeft ten hoogste 1 oplossing.

```
| ?-name(taal,L).
L=[116,97,97,108]
yes
| ?-name(Atoom,[116,97,97,108]).
Atoom=taal
yes
| ?-name(taal,[116,97,97,108]).
yes
| ?-name(mens(socrates),L).
Illegal argument supplied            /* Argument is structuur*/
```

```
| ?-name('mens(socrates)',L).          /* Atoom door de apostrophen */
L=[109,101,110,115,40,115,111,99,114,97,116,101,115,41]
yes
| ?-name(23348,L).
L=[50,51,51,52,56]
yes
| ?-name(123456,L).                    /* Getal is te groot, wordt negatief */
L=[45,55,54,49,54]
yes
| ?-name(Getal,[45,55,54,49,54]). /* We gebruiken 'name' om vorig */
Getal=-7616                            /* resultaat te bekijken      */
yes
```

8.4.2 "Strings"

Aangezien het erg vervelend is een rij van symbolen als een lijst van getallen te moeten noteren, is er in Prolog een speciale notatie voor lijsten van ASCII-codes, de *stringnotatie*. Een lijst van ASCII-codes mag genoteerd worden als de met die codes corresponderende reeks van tekens tussen aanhalingstekens (niet te verwarren met apostrophen!). Bijvoorbeeld: *"abc"* is identiek met *[97,98,99]*: Elk teken tussen de aanhalingstekens wordt als een symbool voor een ASCII code beschouwd, waardoor variabelen niet in een string voor kunnen komen.

```
| ?-"abc"=[97,98,99].
yes
| ?-""=[].                             /* Lege string */
yes
| ?-" "=[32].                          /* 1 spatie in string! */
yes
| ?-name(C,"abc").
C=abc
yes
| ?-name(stress,"stress").
yes
| ?-name(a,"A").                       /* A is hier geen variabele! */
no
```

De twee notatiewijzen moeten wel goed uit elkaar worden gehouden, zoals hieronder blijkt:

```
| ?-"abc"=[a,b,c].                     /* tekens i.p.v. ASCII-codes in lijst */
no
| ?-"a,b,c"=[97,98,99].                /* komma's onbedoeld in stringnotatie */
no
| ?-"a,b,c"=[97,44,98,44,99].          /* komma's mógen in een string, maar */
yes                                    /* dan krijg je wel een andere lijst! */
```

8.4.3 length/2

Een predikaat dat algemeen op lijsten toepasbaar is, en dat vooral gebruikt wordt indien er met *strings* gewerkt wordt, is **length/2**.

length(Lijst,L) unificeert *L* met het aantal elementen van de lijst waaraan *Lijst* gebonden *moet* zijn. **length** geeft hoogstens één oplossing.

```
| ?-length([a,b,c],L).
L=3;                                  /* ; geeft geen andere oplossing! */
no
| ?-length([97,98,99],L).
L=3
yes
| ?-length("abc",L).
L=3
yes
| ?-length("a,b,c",L).
L=5                                   /* !!! */
yes
| ?-length("",L).
L=0
yes
| ?-length(" ",L).
L=1
yes
| ?-length(" ",1).
yes
| ?-length(abc,L).
Illegal argument supplied             /* Geen lijst ! */
| ?-length(L,3).
Illegal argument supplied             /* Er wordt geen lijst van 3 geconstrueerd */
```

8.4.4 Opgaven

8.8 Definieer (in de file *vervoeg.pro*) een predikaat **ott_3e**/2, dat de *onvoltooid tegen - woordige tijd 3e persoon enkelvoud* berekent van een regelmatig werkwoord. Zorg ervoor dat op zijn minst de volgende typen werkwoorden correct behandeld worden:

```
| ?-ott_3e(lopen,X).
X=loopt
yes
| ?-ott_3e(fluiten,X).
X=fluit
yes
| ?-ott_3e(dokken,X).
X=dokt
yes
| ?-ott_3e(aaien,X).
X=aait
yes
```

8.5 De input- en de outputstream

In de bespreking van de I/O predikaten tot nu toe hebben we alleen vánhet toetsenbord, en náár het scherm geschreven. We kunnen ook gegevens van en naar andere I/O apparaten transporteren, in het bijzonder van en naar files op disk. Om dit mogelijk te maken leest de interpretator altijd van een *virtuele* (d.w.z. niet echt bestaande) *inputfile*, de *inputstream*, en schrijft de interpretator altijd naar een *virtuele outputfile*, de *outputstream*. Beide *streams* zijn gekoppeld aan echt bestaande randapparaten. Als de interpretator wordt opgestart, zijn zowel de inputstream als de outputstream gekoppeld aan *user*. Zoals al eerder opgemerkt werd, betekent

dit dat de inputstream gekoppeld wordt aan het *toetsenbord* en de outputstream aan het *scherm*. Er bestaan systeempredikaten, waarmee deze koppelingen gewijzigd kunnen worden. Deze wijzigingen kunnen net zo vaak plaatsvinden als nodig is. Zo is het mogelijk om eerst van de ene inputfile te lezen, te switchen naar een andere inputfile en daarvan te lezen, en vervolgens weer verder te lezen van de eerste.

Er zijn zes predikaten die een stream als argument nemen, drie voor de *inputstream* en drie voor de *outputstream*.

8.5.1 Outputstreams: telling/1, tell/1, told/0

De predikaten die een outputstream als argument nemen zijn **telling/1**, **tell/1** en **told/0**. Een outputstream kan gelijk zijn aan ***user***, aan een *filenaam* of aan implementatieafhankelijke bijzondere namen (bij implementaties voor MS-DOS zijn veelal mogelijk ***con*** (voor *console*), of ***prn*** (voor *printer*); soms is het mogelijk een stream te koppelen aan een *window*; voor al deze mogelijkheden zij men verwezen naar de manual van de gebruikte interpretator).

telling(*Out*) geeft de naam van de huidige outputstream: ***Out*** wordt geünificeerd met de aanduiding voor het fysieke apparaat dat aan de outputstream gekoppeld is. ***Out*** kan een variabele of een atoom zijn.

tell(*Out*) verandert de huidige outputstream in ***Out***. ***Out*** moet een atoom of een daaraan gebonden variabele zijn. Als ***Out*** al eerder als outputstream in gebruik was en niet afgesloten werd door **told** (zie hierna), dan wordt nieuwe output *toegevoegd* aan ***Out***. Was ***Out*** nog niet in gebruik (nog niet *geopend*), dan wordt ***Out*** door **tell** *gecreëerd*.

told sluit de huidige outputstream af, mits deze ongelijk is aan ***user***, en koppelt de outputstream aan ***user***.

In het volgende voorbeeld wordt de output van het predikaat **hallo** van opgave *8.1* naar de file *hallo.txt* gedirigeerd, waardoor we niets op het scherm zien verschijnen.

```
| ?- tell('hallo.txt'),
     hallo(5,20),
     told,
     telling(Who).
Who=user
yes
```

8.5.2 Inputstreams: seeing/1, see/1, seen/0

De predikaten die een outputstream als argument nemen zijn **seeing/1**, **see/1** en **seen/0**.

seeing(*In*) geeft de naam van de huidige inputstream: ***In*** wordt geünificeerd met de aanduiding voor het fysieke apparaat dat aan de inputstream gekoppeld is. ***In*** kan een variabele of een atoom zijn.

see(*In*) verandert de huidige inputstream in ***In***: Het argument moet een atoom of een gebonden variabele zijn die gelijk is aan de naam van een *bestaande* fysieke file. Als ***In*** al eerder als inputstream in gebruik was en niet afgesloten werd door **seen** (zie hierna), dan wordt verder gelezen vanaf het punt waar de vorige keer gestopt was met lezen. Was ***In*** nog niet in gebruik (nog niet *geopend*), dan wordt ***In*** door see geopend.

seen sluit de huidige inputstream af, mits deze ongelijk is aan *user*, en koppelt de inputstream aan *user*.

Als voorbeeld gebruiken we see en seen om **copy_file/2** te definiëren, dat met behulp van *character I/O* een stream naar een andere kopieert:

```
copy_file(From,To):-
        seeing(In)              /* Onthoud huidige inputstream   */
        telling(Out)            /* Onthoud huidige outputstream  */
        see(From),              /* Wijzig Inputstream            */
        tell(To),               /* Wijzig Outputstream           */
        repeat,                 /* Begin van cyclus              */
                get0(C),
                (
                        C=26,   /* ^Z=End_of_file:               */
                        !,      /* Niet meer backtracken         */
                        seen,   /* Afsluiten inputstream         */
                        told,   /* Afsluiten outputstream        */
                        see(In),   /* Terug naar vorige inputstream */
                        tell(Out)  /* Terug naar oude outputstream  */
                ;
                        put(C), /* Schrijf C naar Outputstream   */
                        fail    /* Backtracken (tot aan repeat)  */
                ).
```

We lezen de in de vorige sectie gemaakte file *hallo.txt* weer in met behulp van het predikaat **copy-file**:

```
| ?-copy_file('hallo.txt',user).
        'Hallo!'
        'Hallo!'
        'Hallo!'
        'Hallo!'
        'Hallo!'
yes
```

Hieronder volgen nog wat demo's van **copy_file**:

```
| ?-copy_file(user,'demo.txt').                 /* van scherm -> file */
|: we schrijven nu
|: van het scherm naar een file.
|: Na elke return komt de prompt op het scherm.
|: En dan nu ^Z voor 'end-of-file':
|:^Z
yes
| ?-copy_file('demo.txt',user).                 /* Omgekeerde richting! */
we schrijven nu
van het scherm naar een file.
Na elke return komt de prompt op het scherm.
En dan nu ^Z voor 'end-of-file':
yes
```

8.5.3 Opgaven

8.9 Schrijf (in de file *zinnetje.pro*) een programma om interactief een file met definities voor zelfstandige naamwoorden te vullen. De gemaakte file moet ge(**re**)**consult**eerd kunnen worden. Het te vervaardigen programma, laten we het **woordenboek** dopen, moet zo gebruikersvriendelijk mogelijk zijn Dat wil zeggen dat de gebruiker zoveel mogelijk alleen maar *j* of *n* hoeft te antwoorden op vragen. Voordat de definitie naar een file wordt weggeschreven, krijgt de gebruiker de gelegenheid de definitie te controleren. Hieronder volgt een voorbeeldsessie:

```
| ?-woordenboek.
OUTPUTFILE (tussen apostrophen, afsluiten met PUNT): 'woorden.pro'.
----
Geef WOORD (afsluiten met PUNT) of ^Z |: aap.
        GETAL
                enkelvoud |: j
        LIDWOORD
                de |: j
WOORDDEFINITIE: zelfstandig_naamwoord(aap,enkelvoud,de).
        OKEE? |: j
        Woorddefinitie weggeschreven.
----
Geef WOORD (afsluiten met PUNT) of ^Z |: nootje
        GETAL
                enkelvoud |: n
        LIDWOORD
                de |: n
WOORDDEFINITIE: zelfstandig_naamwoord(nootje,meervoud,het).
        OKEE? |: n
        Woorddefinitie NIET weggeschreven.
----
Geef WOORD (afsluiten met PUNT) of ^Z |: nootje
        GETAL
                enkelvoud |: j
        LIDWOORD
                de |: n
WOORDDEFINITIE: zelfstandig_naamwoord(nootje,enkelvoud,het).
        OKEE? |: j
        Woorddefinitie weggeschreven.
----
Geef WOORD (afsluiten met PUNT) of ^Z |: ^Z
yes
| ?-copy_file('woorden.pro',user).
zelfstandig_naamwoord(aap,enkelvoud,de).
zelfstandig_naamwoord(nootje,enkelvoud,het).
yes
```

Advies: Maak niet één lange clausule, maar splits het probleem op.

9 Metapredikaten

In dit hoofdstuk komen systeempredikaten en enige operatoren aan bod, waarmee een Prologprogramma zichzelf kan bekijken en wijzigen. Omdat deze predikaten (delen van) clausules als argument nemen, noemen we deze operatoren en predikaten ook wel metapredikaten.

9.1 Testpredikaten

We beginnen met de predikaten die onderzoeken of een term in een bepaalde categorie valt. Voor al deze predikaten geldt dat ze ten hoogste één keer slagen. Bij backtracken geven ze niet opnieuw een resultaat.

integer(*T*)	onderzoekt of *T* een geheel getal is.
atom(*T*)	onderzoekt of *T* een atoom (niet-numerieke constante) is.
atomic(*T*)	onderzoekt of geldt **integer(*T*)** of **atom(*T*)**.
var(*T*)	onderzoekt of *T* een variabele is.
nonvar(*T*)	onderzoekt of *T* iets anders dan een variabele is.

Voorbeelden:

```
| ?-integer(-123).
yes
| ?-integer(_123).
no
| ?-integer(pi).
no
| ?-integer('123').
no

| ?-atom(pi).
yes
| ?-atom([]).                    /* De lege lijst is een atoom ! */
yes
| ?-atom(':-').
yes
| ?-atom(restaurant("Cou Tin Ho")).
no
| ?-atom([[]]).
no
| ?-atom(-123).
no
```

```
| ?-atomic('HIV-1').
yes
| ?-atomic(-123).
yes
| ?-atomic(_123).
no

| ?-var(_123).
yes
| ?-var(poes(tom)).
no
```

Uit de volgende voorbeelden blijkt dat **nonvar** bij slagen weinig informatief is over wat de geteste term dan wél is.

```
| ?-nonvar(poes(X)).            /* structuur (met variabele) */
yes
| ?-nonvar([H1,H2|T]).          /* lijst (met variabelen) */
yes
| ?-nonvar(123).                /* getal */
yes
| ?-nonvar(_).                  /* variabele */
no
```

9.1.1 Opgaven

9.1 Definieer een predikaat **term_test/1** dat alle bovengenoemde tests uitvoert op het meegegeven argument en het resultaat van elke test netjes afdrukt. Probeer het predikaat uit op een scala van termen, bijvoorbeeld:

```
| ?-term_test([]).
integer([])     no
atom([])        yes
atomic([])      yes
var([])         no
nonvar([])      yes
yes
```

9.2 Schrijf een predikaat **concat/3** dat twee atomen aan elkaar koppelt.
Bijvoorbeeld:

```
| ?-concat(schotel,antenne,R).
R=schotelantenne
yes
```

Als de concatenatie en één van beide atomen gegeven is, moet het andere atoom bepaald worden, zoals in:

```
| ?-concat(A1,antenne,schotelantenne),
A1=schotel
yes
```

Andere gevallen moeten netjes in **fail** resulteren en niet in een foutmelding (zoals *illegal argument supplied*) van de interpretator. (Voor de eenvoud mag er van worden uitgegaan dat voor het derde argument nooit een structuur wordt opgegeven.)

9.2 Vergelijkingsoperatoren voor termen

Zoals we in hoofdstuk 7 gezien hebben, zijn *getallen* numeriek geordend. Prolog kent nog een andere ordening, namelijk voor *termen* in het algemeen. Termen uit verschillende klassen zijn als volgt geordend:

variabelen < getallen < atomen < structuren

In deze ordening ontbreken *lijsten*, maar in 9.3.3 zal onthuld worden dat een lijst ook een structuur is.
Binnen de diverse klassen van termen gelden de volgende regels.

Variabelen kunnen onderscheiden worden van elkaar, maar ze zijn onderling niet geordend.

Getallen zijn geordend op hun numerieke waarde.

Atomen zijn alfabetisch (*lexicografisch*) geordend, waarbij de individuele tekens geordend zijn volgens hun ASCII-code.

Structuren zijn geordend op drie kenmerken van afnemende belangrijkheid: talligheid, functor, argumenten.

Termen kunnen vergeleken worden met behulp van de volgende *lexicografische vergelijkingsoperatoren* (@ herinnert aan *alfabetisch*):

T1 @< *T2*	onderzoekt of T1 kleiner is dan *T2*
T1 @> *T2*	onderzoekt of T1 groter is dan *T2*
T1 == *T2*	onderzoekt of T1 identiek is met *T2*
T1 \== *T2*	onderzoekt of T1 niet identiek is met *T2*
T1 @<= *T2*	onderzoekt of T1 kleiner of gelijk is aan *T2*
T1 @>= *T2*	onderzoekt of T1 groter of gelijk is aan *T2*

De argumenten van de vergelijkingsoperatoren worden altijd *letterlijk* genomen; er vindt dus nooit unificatie of evaluatie van de argumenten plaats. Hierdoor kunnen we bijvoorbeeld de *namen* van variabelen vergelijken:

```
| ?-A==B.
no
| ?-A==A.
yes
| ?-A=B.                    /* Unificatie ! */
A=B,
B=B
yes
```

Andere voorbeelden zien we hieronder:

```
| ?-aap@<noot.
yes
| ?-aap@>Var.
yes
| ?-dier(aap)\==dier(aap).
no
| ?-zombie(jan)@<dier(orang,oetan).        /* Talligheden ongelijk */
yes
```

Vele implementaties kennen ook het predikaat **compare/3**, dat van twee termen met een symbool aangeeft of de eerste term lexicografisch *kleiner dan* (<), *groter dan* (>) of *gelijk aan* (=) de tweede term is:

```
| ?-compare(LexRel,Var,boom(baobab)).
LexRel=<
yes
```

9.3 Analyse en synthese van structuren

Structuren kunnen ontleed en geconstrueerd worden met behulp van de predikaten **arg** en **functor**.

9.3.1 Arg

Met arg/3 is het mogelijk met behulp van een plaatsaanduiding een argument uit een structuur te plukken. Het omgekeerde, het toekennen van een waarde aan een argument, is ook mogelijk:

arg(***Plaats***,***Struct***,***Argum***): ***Argum*** wordt geünificeerd met het argument op plaats ***Plaats*** (een geheel getal) van de structuur ***Struct***. ***Plaats*** en ***Struct*** mogen geen variabele zijn. (***Struct*** mag wel variabelen bevatten). Voorbeelden zien we hieronder:

```
| ?- Struct=broer(jan,piet),
     arg(1,Struct,X),                /* selectie van */
     arg(2,Struct,Y).                /* argumenten */
X=jan,
Y=piet
yes

| ?- Struct=broer(X,Y),
     arg(1,Struct,jan),              /*Binden van */
     arg(2,Struct,piet).             /* argumenten   */
X=jan,
Y=piet,
Struct=broer(jan,piet)
yes
```

```
| ?- Struct=broer(X,piet),                /* Gecombineerd: */
     X=jan,
     Y=piet,
     Struct=broer(jan,piet).
yes

| ?-arg(2,broer(_1,_2),X).
_2= X                                     /* X en _2 zijn geünificeerd. */
X= X                                      /* Voor X en _1 zijn geen      */
_1= _1                                    /* oplossingen. */
yes

| ?-arg(2,broer(jan,piet),jan).           /* testen van waarde */
no

| ?-arg(3,broer(jan,piet),X).             /* Een plaats te ver */
no

| ?-arg(2,Struct,piet).
Illegal argument supplied

| ?-arg(1,2+3+4,X)./* Operator */
X=2+3/* want: 2+3+4 == +(+(2,3),4) */
yes
| ?-arg(2,2+3+4,X).
X=4
yes
```

9.3.2 Functor

Een predikaat dat niet zozeer op de argumenten opereert, maar waarbij de functor en de talligheid, dus de meer globale kenmerken van een structuur centraal staan is het predikaat **functor/3**. De werking van **functor(Struct,Func,Tal)** hangt als volgt af van het eerste argument (***Struct***).
(a) Is het eerste argument (***Struct***) gebonden aan een structuur, dan worden de twee overige argumenten geünificeerd met respectievelijk de functor (***Func***) en het aantal argumenten (***Tal***: de "talligheid") van de structuur.
(b) Geldt **var(Struct)**, dan moeten de twee overige argumenten constanten zijn en wordt het eerste argument (***Struct***) geünificeerd met een structuur waarvan de functor ***Func*** en het aantal argumenten ***Tal*** is. Hierbij wordt voor elk van de argumenten een variabele ingevuld door de interpretator. Er wordt in dit geval dus een structuur *geconstrueerd*.

Voorbeelden:

```
| ?-functor(poes(tom),F,N).
F=poes,
N=1
yes
| ?-functor(poes(tom),poes,1).
yes
| ?-functor(Term,poes,1).                 /* De interpretator verzint */
Term=poes(A_1)                            /* zelf namen voor variabelen */
yes
```

```
| ?-functor(Struct,kind,2).
Struct=kind(A_1,B_1)                    /* Zie vorige voorbeeld */
yes

| ?-functor(a+3*2,F,N).
F=+,                                    /* Operatoren zijn ook functoren */
N=2
yes
```

In het volgende predikaat (**vraag_kind_ouder**/1) wordt geïllustreerd, hoe met behulp van **functor** een propositie kan worden samengesteld uit termen die ingelezen worden.

```
vraag_kind_ouder(Prop):-                    /* Kan beter, zie opgave 9.3 */
    functor(Prop,kind,2),                   /* Prop heeft vorm: kind(K,O) */
    write('Kind '),read(K),arg(1,Prop,K),   /*1e argument */
    write('Ouder'),read(O),arg(2,Prop,O).   /* 2e argument */
```

Voorbeeld van een dialoogje:

```
| ?-vraag_kind_ouder(P).
Kind |: willem_alexander.
Ouder|: claus.
P= kind(willem_alexander,claus)
yes
```

De mogelijkheid een propositie te construeren zoals in het voorbeeld hierboven is zinnig, omdat zo'n door een programma geconstrueerde propositie, zoals we verderop zullen zien, als doelpropositie aan de interpretator kan worden aangeboden. Bovendien kunnen proposities (en ook regels) door een programma aan de gegevensbank worden toegevoegd of daaruit worden verwijderd. (Proposities kunnen overigens ook geconstrueerd worden met behulp van ***univ*** .)

9.3.3 De functor van een lijst

Tot nu toe hebben we lijsten altijd genoteerd als twee rechte haakjes met daartussen de argumenten. Dit is slechts een door de Prologinterpretator toegestane handige schrijfwijze, die we ook zullen blijven hanteren; in werkelijkheid echter is een lijst een structuur, opgebouwd met behulp van een *tweeplaatsige functor*. Deze functor, ook wel ***cons*** genoemd, wordt voorgesteld door een punt (*dot*) '.'. In het eerste argument van de functor staat de ***kop***, in het tweede argument de ***staart***. Het einde van een lijst wordt aangegeven met de lege lijst, het atoom *[]*. Bijvoorbeeld de lijst die we altijd genoteerd hebben als

```
[a,B,c]
```

is intern gerepresenteerd als:

```
.(a,.(B,.(c,[])))
```

De unificatieoperator heeft dan ook geen moeite met het unificeren van de twee termen:

```
| ?-[a,B,c]='.'(a,'.'(B,'.'(c,[]))).
B=B
yes
```

Zelfs de identiteitsoperator, die alleen maar slaagt als twee termen, inclusief de namen van variabelen, volledig identiek zijn, geeft 'yes' als we de twee notationele varianten lexicografisch vergelijken:

```
| ?-[a,B,c]  ==  '.'(a,'.'(B,'.'(c,[]))).
yes
```

De lijstbouwende functor komt te voorschijn als we lijsten analyseren. Bijvoorbeeld als we **functor** en **arg** op de lijst *[a,b,c]* toepassen:

```
| ?-functor([a,b,c],Func,N).
Func= .,                          /* De functor */
N= 2                              /* De talligheid */
yes
| ?-arg(1,[a,b,c],L).             /* De kop */
L= a
yes
| ?-arg(2,[a,b,c],L).             /* De staart */
L= [b,c]
yes
| ?-arg(3,[a,b,c],L).
no                                /* De '.' is 2-plaatsig! */
```

9.3.4 Opgaven

9.3 Het hierboven beschreven predikaat **vraag_kind_ouder** werkt niet alleen met een variabele als argument, maar ook wanneer de te construeren propositie geheel of gedeeltelijk ingevuld wordt meegegeven. Storend is dat ook bekende argumenten worden opgevraagd.

```
| ?-vraag_kind_ouder(kind(willem_alexander,O)).
Kind |: willem_alexander
Ouder|: claus
O=claus
yes
```

Verander **vraag_kind_ouder** zó, dat in het argument ook de propositie al of niet met variabelen *mag* staan en dat alleen onbekende argumenten worden opgevraagd. Bijvoorbeeld:

```
| ?-vraag_kind_ouder(kind(Willem_alexander,O)).
Ouder|: claus.
O=claus
yes
| ?-vraag_kind_ouder(kind(K,claus)).
Kind |: willem_alexander.
K=willem_alexander
yes
| ?-vraag_kind_ouder(kind(K,O)).
Kind |: willem_alexander.
Ouder|: claus.
K=willem_alexander,
O=claus
yes
```

```
| ?-vraag_kind_ouder(kind(willem_alexander,claus)).
yes

| ?-vraag_kind_ouder(kind(KO)).          /* geen komma */
no
| ?-vraag_kind_ouder(zoon(willem_alexander,claus)).
no
```

9.4 Univicatie (=..) van structuur en lijst

Een structuur kan in een lijst omgezet worden en omgekeerd, door middel van de *univ*-operator (=..).

De doelpropositie **Struct =.. Lijst** heeft tot gevolg dat de functor van de structuur ***Struct*** geünificeerd wordt met de kop van de lijst ***Lijst*** en de overige elementen met de staart van ***Lijst***.

Voorbeelden:

```
| ?-broer(jan,piet) =.. L.
L=[broer,jan,piet]
yes
| ?-S =.. [broer,jan,piet].
S=broer(jan,piet)
yes
| ?-broer(X,Y) =.. [P,jan,Z].
X=jan,
Y=Z,
Z=Z,
P=broer
yes
| ?-functor(broer(jan,piet),broer,2) =.. L.
L=[functor,broer(jan,piet),broer,2]
yes
| ?-3+2*4 =.. L.
L=[+,3,2*4]
yes
| ?-aap =.. X.
X=[aap]
yes
| ?-X =.. [aap].
X=aap
yes
| ?-X =.. Z.
Illegal argument supplied
```

De term links van de *univ*-operator wordt altijd opgevat als een structuur. We kunnen ook een lijst analyseren. We zien dan weer de *punt* als functor opduiken.

```
| ?-[a,b,c] =.. L.
L=[.,a,[b,c]]
```

```
yes
| ?-[a,b] =.. L.
L=[.,a,[b]]
yes
| ?-[a] =.. L.
L=[.,a,[]]
yes
| ?-[] =.. L.
L=[[]]
yes
```

9.4.1 Opgaven

9.4 Definieer **vraag_kind_ouder** met behulp van de univ-operator in plaats van **functor** en **arg** te gebruiken.

9.5 Schrijf een (recursief) predikaat **novars** dat nagaat of er in een willekeurige term variabelen voorkomen. (Een term die geen variabelen bevat heet een *ground term.*) Voorbeelden:

```
| ?-novars(X).
no
| ?-novars(2).
yes
| ?-novars(poes(tom)).
yes
| ?-novars(poes(Tom)).
no
| ?-novars([a,b,c]).
yes
| ?-novars([a,_,c]).
no
| ?-novars([poes(tom),mens(vrouw(X))])
no
```

9.5 call/1 en *metavariabele*

Naast mogelijkheden om aan proposities te sleutelen, ligt het voor de hand dat het ook mogelijk moet zijn een tijdens een programma geconstrueerde propositie tot doelpropositie te maken. Welnu dit kan door middel van **call/1** of een *metavariabele.*

Is de doelpropositie **call(*Vraag*)** dan wordt de al dan niet samengestelde vraag ***Vraag*** op afleidbaarheid onderzocht. In vele implementaties van Prolog is het ook mogelijk een variabele die gebonden is aan een vraag, als conjunct of disjunct in een vraag op te nemen. Een variabele die op deze manier gebruikt wordt noemen we een *metavariabele.*

In het volgende predikaat, **bekend_ko/0**, wordt eerst met **vraag_kind_ouder** een propositie geconstrueerd (***P***), waarvan vervolgens de afleidbaarheid nagegaan wordt; met andere woorden: er wordt onderzocht of de geconstrueerde *kind_ouder relatie* in de gegevensbank staat:

```
bekend_ko:-
      vraag_kind_ouder(P),
      call(P).
```

Beschikken we over een gegevensbank met daarin de stamboom van het koningshuis, dan is een mogelijke dialoog:

```
| ?-bekend_ko.
Kind |: willem_alexander.
Ouder|: claus.
yes.                 /* want kind(willem_alexander,claus) is afleidbaar */
| ?-bekend_ko.
Kind |: hugo.
Ouder|: claus.
no                   /* want kind(hugo,claus) is niet afleidbaar */
```

9.6 not/1

Met behulp van **call** kunnen we *negatie* invoeren op de manier waarop dat in alle standaardversies van Prolog gebeurt, namelijk als het mislukken van de poging de afleidbaarheid van een propositie aan te tonen (*negation as failure*). Het resultaat van **not(P)**, waarin ***P*** een propositie is, is omgekeerd aan dat van P. Een definitie van **not** met behulp van een disjunctie en **true** is de volgende:

```
not(P) :- call(P), !, fail; true.
```

P wordt door de **call** op afleidbaarheid onderzocht: levert dat een succes op, dan moet **not(P)** als resultaat **fail** opleveren, hetgeen bereikt wordt door middel van de **fail**; de **cut** voorkomt dat we door backtracking in de tweede tak van de disjunctie terecht komen. Is **call(P)** niet afleidbaar, dan komen we wel in de tweede tak en daarin staat **true**, waardoor **not(P)** slaagt. We kunnen met **not** bijvoorbeeld **onbekend_ko** als volgt definiëren:

```
onbekend_ko :- not(bekend_ko).
```

9.7 asserta/1, assertz/1, assert/1

Er zijn drie systeempredikaten waarmee clausules aan de gegevensbank kunnen worden toegevoegd.

asserta((Head:-Body)) breidt de definitie van het predikaat ***Head*** in de gegevensbank uit door de clausule **Head:-Body** vóór de al aanwezige clausules van de definitie van *Head* te zetten. De *a* achteraan in **asserta** is een geheugensteuntje: zoals de *a* de eerste letter is in het alfabet, zo is de nieuwe clausule de eerste in de gewijzigde definitie van ***Head***.

assertz((Head:-Body)) breidt de definitie van het predikaat ***Head*** in de gegevensbank uit door de clausule **Head:-Body** áchter de al aanwezige clausules van de definitie van ***Head*** te zetten. De toegevoegde clausule wordt dus de laatste in de definitie van ***Head***, vandaar assertz.

assert((Head:-Body)) breidt de definitie van het predikaat ***Head*** in de gegevensbank uit door de clausule **Head:-Body** toe te voegen aan de definitie van ***Head***. De plaats in het rijtje van clausules doet niet ter zake.

Als een clausule moet worden toegevoegd zonder ***Body***, dan mogen de haakjes rondom de clausule, het ***indien-teken*** en de ***Body*** weggelaten worden. Een andere mogelijkheid is om in dat geval voor ***Body*** **true** in te vullen.

In het volgende voorbeeld wordt het predikaat **voeg_toe_ko** gedefinieerd dat een *kind_ouder relatie* aan de gegevensbank toevoegt, mits deze zich nog niet in de gegevensbank bevindt:

```
voeg_toe_ko:-
        vraag_kind_ouder(P),
        (
                not(P), !,               /* P nog niet in gegevensbank */
                assertz(P),              /* Zet P achteraan */
                write('Toegevoegd')      /* Meld toevoeging */
                ;
                write('Al aanwezig!')    /* Meld aanwezigheid */
        ).
```

9.8 retract/1 en abolish/2

Het verwijderen van clausules kan met behulp van **retract/1** en **abolish/2**.

retract((***Head***:-***Body***)) verwijdert de eerste clausule uit de gegevensbank die unificeerbaar is met het meegeven argument.
Head moet (aan) een atoom of structuur (gebonden) zijn. Bij backtracken kan een volgende clausule verwijderd worden.

abolish(*Pred,Talligheid*) verwijdert alle clausules (feiten én regels) waarvan het hoofd een propositie van de vorm ***Pred/Talligheid*** is. ***Pred*** moet (aan) een atoom (gebonden) zijn, ***Talligheid*** moet een geheel getal zijn. Dit predikaat slaagt altijd.

In het volgende voorbeeld wordt een predikaat gedefinieerd dat een of meer *kind_ouder relaties* uit de gegevensbank verwijdert. Er wordt een waarschuwing gegeven als het te verwijderen feit zich niet in de gegevensbank bevindt. Meer dan één feit kan verwijderd worden als er voor het kind en/of de ouder een variabele wordt opgegeven:

```
verwijder_ko:-
        vraag_kind_ouder(P),!,
        (
                P,                  /* In P variabelen -> meer oplossingen
                mogelijk */
                retract(P),         /* In P zijn alle variabelen gebonden! */
                write('Verwijderd'),nl,
                fail                /* Forceer herhaling */
        ;
                write('Geen (meer) aanwezig!')
        ).
```

9.9 clause/2

Met het predikaat **clause/2** is het mogelijk om te onderzoeken of een clausule zich in de gegevensbank bevindt.

clause(Head,Body) zoekt een clausule waarvan het hoofd unificeerbaar is met ***Head***, waarbij ***Head*** (aan) een atoom of structuur (gebonden) moet zijn. De proposities rechts van het indien-teken in de gevonden clausule worden geünificeerd met ***Body***. Als de clausule een feit is, dan wordt ***Body*** geünificeerd met **true**. Door backtracking kunnen meerdere clausules gevonden worden.

Als voorbeeld definiëren we het predikaat **toon_ko** dat alle *kind-ouder relaties* in de gegevensbank op het scherm laat zien:

```
toon_ko:-
        clause(kind(K,O),true),
        write(kind(K,O)),nl,
        fail.
toon_ko.
```

9.10 Regels als argument

Vanwege het bijzondere belang van het *indien-teken* wordt de interpretator (en kort daarop meestal ook de programmeur) erg nerveus, wanneer een argument van een propositie bestaat uit een term waarin het *indien-teken* voorkomt, zónder dat er haakjes om die term staan. We zullen hieronder een voorbeeld laten zien waarin, in tegenstelling tot de in voorgaande voorbeelden, *regels* gemanipuleerd worden. We proberen eerst een clausule vanaf ***user*** te definiëren, wat pas bij de tweede poging lukt:

```
| ?-consult(user).
voegtoe(Head:-Body):-                     /* Haakjes vergeten!*/
        assertz(Head:-Body),              /* Idem!*/
        write('Toegevoegd!'),nl.
Syntax error                              /* Geweigerd */
voegtoe((Head:-Body)):-                   /* Nu goed */
        assertz((Head:-Body)),
        write('Toegevoegd!'),nl.
yes
^Z
```

Nu gebruiken we het zojuist gedefinieerde **voegtoe** (een "pratende" assert).

```
| ?-voegtoe(a:-b).                        /* Haakjes vergeten*/
Syntax error
| ?-voegtoe((a:-b,c,d)).
Toegevoegd!
yes
| ?-clause(a,B),retract(a:-B).            /* Geen haakjes!*/
Syntax error
| ?-clause(a,B),retract((a:-B)).
B=b,c,d
yes
```

9.11 Opgaven

9.6 In de loop van dit hoofdstuk werden bij wijze van voorbeeld enige predikaten gedefinieerd waarbij *kind_ouder relaties* gemanipuleerd werden. Bij elkaar vormen die predikaten een database-systeempje. Hieronder staat een overkoepelend programma **database_ko**/0, dat het mogelijk maakt de diverse operaties via een *menu* uit te voeren. Probeer dit eens uit (zet de definities in de file *database.pro*). Er staan in het menu een paar operaties, waarvan de predikaten nog niet gedefinieerd zijn, nl. **vervang_ko** en **file_ko**. Definieer deze predikaten.

```
database_ko:-
        repeat,
        write(                          /* Nu komt één atoom */
'

                Kind_ouder MENU:
                ---------------
                1=toevoegen
                2=verwijderen
                3=vervangen

                8=toon alle kind_ouder relaties op scherm
                9=bewaar alle kind_ouder relaties in een file
                0=stoppen
                ---------------'),
        vraag_keuze(Keuze),
        (
                stop(Keuze),!
        ;
                doe(Keuze),
                fail
        ).

vraag_keuze(Keuze):-
        nl,
        write('Keuze '),
        get(Keuze),
        skip(31).

stop(48):-
        nl, write('Tot Ziens!'), nl.

doe(49):-!,write('TOEVOEGING:'),nl,voeg_toe_ko.
doe(50):-!,write('VERWIJDERING:'),nl,verwijder_ko.
doe(51):-!,write('VERVANGING:'),nl,vervang_ko.
doe(56):-!,write('KIND  relaties:'),nl,toon_ko.
doe(57):-!,write('->FILE:'),nl,file_ko.
doe(_):-!,write('***FOUTE  KEUZE!'),nl.

vervang_ko:- write('Kan ik nog niet! ***').

file_ko:- write('Nog niet geimplementeerd! ***').
```

9.7 Corrigeer de verderop gegeven definitie van het predikaat **verzamel_destructief/2**. **verzamel_destructief** moet aan de volgende omschrijving voldoen:
(1) het eerste argument bevat een propositie met 1 variabele als argument;
(2) het tweede argument bevat een variabele die, nadat het predikaat verwerkt is, geünificeerd is met een lijst die alle oplossingen voor de variabele bevat die de meegegeven propositie kan waarmaken op grond van de aanwezigheid van een corresponderend feit in de gegevensbank. Als zijeffect worden alle corresponderende feiten uit de gegevensbank verwijderd.
Staan in de gegevensbank de volgende feiten:

```
is_vrouw(beatrix).
is_vrouw(irene).
is_vrouw(margriet).
is_vrouw(christina).
```

dan kunnen we **verzamel_destructief** als volgt gebruiken om alle vrouwen in een lijst te verzamelen en tegelijkertijd uit de gegevensbank verwijderen (zoals uit de tweede vraag blijkt):

```
| ?-verzamel_destructief(is_vrouw(V),L).
V=beatrix
L=[beatrix,irene,margriet,christina]
yes.
| ?-verzamel_destructief(is_vrouw(X),L).
L=[]
no
```

Zoek uit waarom de volgende definitie (op te nemen in de file *database.pro*) niet goed is en verbeter de fouten:

```
verzamel_destructief(P,[X|L]):-        /* Te verbeteren ! */
        retract(P),!,
        arg(1,P,X),
        P=..[Pred,_],
        verzamel_destructief(P,L).
verzamel_destructief(_,[]).
```

9.8 Definieer het predikaat **vraag_propositie**, dat eerst vraagt om het *predikaat*, dan om het aantal *argumenten*, en dat vervolgens elk van de argumenten opvraagt. De samengestelde propositie wordt geünificeerd met het enige argument. Bijvoorbeeld:

```
| ?-vraag_propositie(Prop).
Predikaat|: ouders.
Aantal argumenten|: 3.
Arg 1|: claus.
Arg 2|: beatrix.
Arg 3|: willem_alexander.
Prop=ouders(claus,beatrix,willem_alexander)
yes
| ?-vraag_propositie(Prop).
Predikaat|: start.
Aantal argumenten|: 0.
Prop=start
yes
```

9.9 In het predikaat **vraag_propositie** uit de vorige opgave werd een propositie gegeconstrueerd waarbij ook het predikaat door de gebruiker kan worden gespecificeerd.
(a) Gebruik dit predikaat om de basisoperaties **voeg_toe_ko, verwijder_ko,** enzovoorts te veralgemeniseren tot predikaten die niet meer uitsluitend feiten van het type ***kind(K,O)*** aankunnen. Herdoop **database_ko** tot **database.**
(b) Neem vervolgens voor elk *type* feit dat in de file *stamboom.pro* voor kan komen expliciet de structuur in de gegevensbank op als *metafeit.* Bijvoorbeeld:

data_structuur(kind(kind,ouder)).

Gebruik deze *metafeiten* om namen aan de argumenten te geven, zodat in de prompts voor de argumenten omschrijvingen (bijvoorbeeld *kind*) verschijnen (in plaats van kreten als *arg 2*). In sommige gevallen kunnen de metafeiten gebruikt worden om te controleren of een bepaald type feit wel in de database toegelaten mag worden.
(c) Als de *metafeiten* ook nog opgevraagd kunnen worden en er voorzieningen getroffen worden om ook regels toe te voegen en te verwijderen, dan zijn we al aardig op weg naar een algemeen database-pakket. Voor de liefhebbers.

9.12 bagof/3 en setof/3

In de meeste Prologimplementaties zijn met behulp van enige van de besproken metapredikaten (met name met **call, assertz** en **retract**) twee predikaten (**bagof**/3 en **setof**/3) gedefinieerd, waarmee alle oplossingen voor één of meer variabelen van een *vraag* verkregen kunnen worden. Met behulp van de gevonden variabelebindingen worden *termen* geconstrueerd die in een *lijst* verzameld worden. Het verschil tussen **bagof** en **setof** is dat **setof** elke term maximaal één keer in de lijst plaatst (*set*=verzameling) en bovendien de elementen van de oplossingenverzameling lexicografisch sorteert; **bagof** daarentegen plaatst élke geconstrueerde term in de lijst (*bag*=zak).

De vorm waarin de predikaten gebruikt worden is als volgt:

bagof(*term(Var1,Var2,...),vraag(Var1,Var2,...),Lijst*)
setof(*term(Var1,Var2,...),vraag(Var1,Var2,...),Lijst*)

Gebruik van deze predikaten is alleen zinvol als de *vraag* in het tweede argument en de *term* in het eerste argument één of meer gemeenschappelijke variabelen hebben. We onderscheiden twee gevallen:

(1) alle variabelen van de *vraag* bevinden zich in de term,
(2) de *vraag* bevat variabelen die niet in de term voorkomen.

ad (1) In dit geval worden alle mogelijke oplossingen voor de variabelen van de vraag gezocht; elke *term* die ontstaat na substitutie van de gevonden variabelebindingen wordt als element in de *lijst* opgenomen.
Uitgaande van de stamboom op bladzijde 42, krijgen we bijvoorbeeld:

```
| ?-bagof(K,kind(K,beatrix),Kinderen).
K=...                                                    /* Irrelevant */
Kinderen=[willem_alexander,johan_friso,constantijn];     /* Puntkomma! */
no
```

```
| ?-bagof(
      beatrix/K,                                    /* Term */
      kind(K,beatrix),                              /* Vraag */
      Kinderen
).                                                  /* Lijst */
K=...                                               /* Irrelevant */
Kinderen=[beatrix/willem_alexander,beatrix/johan_friso,beatrix/constantijn]
yes
```

Hieronder gebruiken we **setof** om een predikaat te definiëren dat de *vereniging* van twee verzamelingen bepaalt:

```
| ?-[user].
union(V1,V2,U):-
      setof(
            Element,                                /* Term */
            (
                  is_elem_van(Element,V1);          /* Vraag bestaande */
                  is_elem_van(Element,V2)           /* uit een disjunctie */
            ),
            U                                       /* Lijst van termen */
      ).
^Z
yes
| ?-Union([a,b,c],[c,e,b,d],U).
U=[a,b,c,d,e]                           /* Elementen uniek en gesorteerd */
yes
```

ad (2) Wanneer de *vraag* een of meer variabelen bevat, die niet in de *term* voorkomen, dan wordt bij elke combinatie van waarden van deze extra variabelen een lijst van termen gezocht, waarbij elke term correspondeert met een oplossing voor de overige (gemeenschappelijke) variabelen. In dit geval kunnen **setof** en **bagof** dus meer dan één oplossing hebben. Gaan we weer uit van de bekende stamboom, dan is de volgende sessie illustratief (maar niet zinnig):

```
| ?-bagof(K,grootouder(G,K),Kleinkinderen)
K=...,
G=juliana,                                              /* Eerste mogelijkheid */
Kleinkinderen=[willem_alexander,johan_friso,constantijn]; /* puntkomma */
K=...,
G=bernhard,                                             /* Tweede mogelijkheid */
Kleinkinderen=[willem_alexander,johan_friso,constantijn]; /* puntkomma */
no
```

Met **bagof** en **setof** moet niet te royaal worden omgesprongen, aangezien dit tot inefficiënte programma's kan leiden. Voor de eerste versie van een oplossing kunnen deze predikaten handig zijn.

9.12.1 Opgave

9.10 Definieer enige van de verzamelingtheoretische operaties met behulp van **setof** of **bagof**.

10 Operatoren

In Prolog is het mogelijk één- of tweeplaatsige functoren en predikaten te *declareren als operator*, waardoor deze extra syntactische mogelijkheden krijgen. Het opvallendste kenmerk daarbij is, dat er door het gebruik van operatoren minder haakjes nodig zijn om termen te noteren (verderop zullen we zien hoe dat beregeld is). Er zijn veel functoren en predikaten die door de interpretator zelf als operator gedeclareerd zijn:

de rekenoperatoren: +, -, *, /,**mod**;
de **is** operator;
de numerieke vergelijkingsoperatoren: <, >,=<, >=, =:=, =/=;
de algemene vergelijkingsoperatoren: @<, @>,@=<, @>=, \==, ==;
de metaoperatoren: =, =.., \=;
de clausulebouwende operatoren: :-, , , ;.

Vaak is ook **not** als operator gedeclareerd. Een ander voorbeeld van een systeemoperator is de herschrijfpijl, **-->**, waarmee grammatica's kunnen worden gedefinieerd. De gebruiksmogelijkheden van deze belangrijke operator zullen in hoofdstuk 11 besproken worden. Het gebruik van operatoren kan een programma leesbaarder maken, bijvoorbeeld doordat het programma daardoor aansluit bij een notationeel systeem uit een of ander vakgebied. Hier zullen we zien wat operatoren precies zijn en hoe we deze ook zelf met behulp van het metapredikaat **op/3** kunnen declareren.

10.1 Operatoren en hun context

Een doelpropositie die genoteerd is met behulp van operatoren, kan altijd teruggebracht worden tot een term bestaande uit een functor met meer of minder complexe argumenten. De "hoogste" functor fungeert als predikaat en bepaalt wat het effect is van de doelpropositie. Alle tot nu toe gebruikte operatoren zijn systeemoperatoren met specifieke gevolgen, op voorwaarde dat ze in de adequate context worden gebruikt: de rekenoperatoren hebben berekeningen tot gevolg als ze gebruikt worden in de context van een **is** operator, clausules worden in de gegevensbank gezet wanneer ze door een **consult** gezien worden, enzovoorts. De speciale effecten van een operator ontbreken wanneer we deze in een andere context gebruiken, hetgeen toegestaan is en vaak de leesbaarheid van een programma ten goede komt.
Zo kunnen we bijvoorbeeld met onszelf afspreken, dat we de taalkundige kenmerken van een woord of constituent met behulp van de operator = als volgt in een lijst zetten:

[***kenmerk_1=waarde_1, kenmerk_2=waarde_2, ...***]

Het symbool = fungeert hier slechts als scheidingsteken om de leesbaarheid van het programma te verhogen en heeft in deze context niets met unificatie te maken. We zien = in zijn nieuwe rol bijvoorbeeld in het predikaat **geef_kenmerk/2**, dat een *kenmerk=waarde*-paar zoekt in een kenmerkenlijstje:

```
geef_kenmerk(F=V,[F=V|Rest]):-  !.
geef_kenmerk(F=V,[_|Rest]) :-  geef_kenmerk(F=V,Rest).
```

Voorbeelden van het gebruik van **geef_kenmerk**:

```
| ?-geef_kenmerk(persoon=P,[persoon=1,getal=enkelvoud]).
P=1
yes
| ?-geef_kenmerk(getal=G,[persoon=1,getal=enkelvoud]).
G=enkelvoud
yes
```

Het gezochte kenmerk moet zich uiteraard wel in de lijst bevinden:

```
| ?-geef_kenmerk(g=G,[persoon=1,getal=enkelvoud]).
no
```

Bovendien moeten we wel de juiste operator gebruiken. Gebruiken we in een doelpropositie bijvoorbeeld het deelteken (/) in plaats van =, dan gaat het mis:

```
| ?-geef_kenmerk(getal/G,[persoon/1,getal/enkelvoud]).
no
```

We hadden ook een ander symbool dan = als scheidingsteken kunnen kiezen, maar het gebruikte symbool moeten wel een ***operator*** zijn. Proberen we **geef_kenmerk** bijvoorbeeld te gebruiken met behulp van het niet als operator gedeclareerde symbool **:=>**, dan komt de interpretator niet eens aan een afleidingspoging toe, omdat de doelpropositie syntactisch niet in orde is:

```
| ?-geef_kenmerk(getal:=>G,[persoon:=>1,getal:=>enkelvoud]).
Syntax error
```

Als we toch graag :=> willen gebruiken, dan moeten we :=> als operator declareren en dat kan met behulp van **op/3** , dat verderop, na de nodige voorbereidende opmerkingen, besproken wordt.

10.2 Aantal argumenten en positie

Zoals reeds opgemerkt werd, kunnen alleen functoren met één of twee argumenten als operator gedeclareerd worden. We noemen deze operatoren respectievelijk *unaire* en *binaire* operatoren.

binair:	**+ - * / mod = < > =< >= =:= =/= =\= is =..**
unair:	**not + -**

Een *binaire* operator is altijd een *infix-operator*, hetgeen wil zeggen dat de operator tussen de twee argumenten moet staan. Een *unaire* operator kan gedeclareerd worden als een *prefix-* of als een *postfix-operator*, wat betekent dat de operator respectievelijk vóór of achter het enige argument staat. Een functor of predikaat kan tegelijkertijd zowel een unaire als een binaire operator zijn. Het plus- en het minteken zijn daar voorbeelden van.

Binair gebruikt:	**N-1, N+1**
Unair gebruikt:	**-1, -N, +1, +N**
Beide:	**-N-1, -(N-1), +N+1, +(N+1)**

Plus- en minteken zijn beide unaire prefixoperatoren. Een voorbeeld van een unaire postfixoperator is 'kwadraat' in uitdrukkingen als:

2 kwadraat
B kwadraat - 4*A*C

Let wel: machtsverheffen bijvoorbeeld, aangegeven met ^ (*dakje*), is binair:

2^2
B^2 - 4*A*C

Operatoren hebben we al vaak in een heel andere context dan die van het rekenen gebruikt, namelijk bij het definiëren van clausules:

indien-operator	:-	(binair (!))
conjunctie-operator	,	(binair)
disjunctie-operator	;	(binair)

Elke operator, dus ook het indien-symbool, kan ook in de functornotatie gebruikt worden. We hadden het eerste Prologprogramma in dit boek:

```
sterfelijk(X) :- mens(X).
mens(socrates).
```

ook zo kunnen noteren:

```
:-(sterfelijk(X),mens(X)).        /* :- binair gebruikt */
:-(mens(socrates),true).          /* :- binair gebruikt */
```

De propositie **true** als tweede argument in de tweede clausule is noodzakelijk, omdat **:-** met één argument iets anders betekent, namelijk een *directief* (zie 2.8.4). **:-** is dus weer een voorbeeld van een operator die zowel binair als unair gebruikt kan worden.

10.3 Prioriteit en associativiteit

Wanneer we een term noteren met behulp van operatoren, ontstaat het probleem welke onderdelen van de term bij elkaar genomen moeten worden als er in de term meerdere operatoren voorkomen. Voor rekenkundige expressies geldt dat machtsverheffen "voorafgaat" aan worteltrekken (in schoolse termen: √ zónder bovenstreep, want deze heeft dezelfde functie als het zetten van haakjes). Worteltrekken gaat op zijn beurt vooraf aan vermenigvuldigen en delen. Deze laatste twee gaan voor op optellen en aftrekken. De Prologinterpretator probeert elke term om te zetten in de normale vorm, welke bestaat uit een functor of predikaat gevolgd door de argumenten tussen haakjes. De manier waarop de onderdelen van een met operatoren samengestelde term samengenomen worden, wordt geregeld door aan elke operator een *prioriteit* en een *associativiteit* toe te kennen.

10.3.1 Prioriteit

Aan elke operator wordt een getal toegekend. Dit getal wordt de *prioriteit* (*precedence*) van de operator genoemd. De prioriteit van een operator moet worden opgevat als een *rangnummer*: de operatoren met het *laagste* nummer hebben voorrang op die met een hoger nummer. Het gevolg hiervan is, dat de operator met het hoogste getal als prioriteit het meeste materiaal van de term

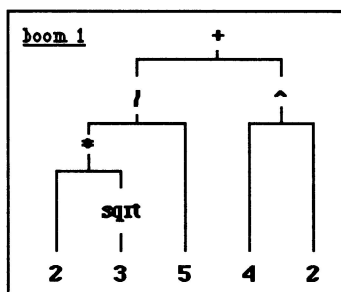
omvat; nog anders gezegd: de operator met het hoogste rangnummer heeft het grootste *bereik*. Bijvoorbeeld de rekenoperatoren hebben in vele Prologinterpretatoren de volgende waarden:

^	Machtsverheffen	200
sqrt	Worteltrekken	250 (unair)
*	Vermenigvuldigen	400
/	Delen	400
+	Optellen	500
-	Aftrekken	500

We kunnen de structuur van een expressie met een boompje weergeven. Zo correspondeert met

2 * sqrt 3 / 5 + 4^2

onderstaande boom (boom 1):



Boom 1 kan ook met functoren weergegeven worden, waarbij de hoogste operator de hoogste functor wordt:

+(/(*(2,sqrt(3)),5),^(4,2))

Bovenstaande term illustreert hoe nuttig operatoren kunnen zijn voor de leesbaarheid van termen. De leesbaarheid kan nog verhoogd worden door gebruik te maken van haakjes (zie hieronder).

10.3.2 Haakjes en normaal genoteerde termen

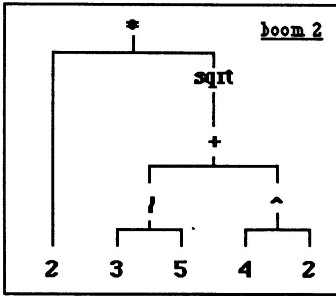
Subtermen tussen haakjes gaan altijd voor op de rest, zoals we gewend zijn voor rekenexpressies. Zo correspondeert bijvoorbeeld

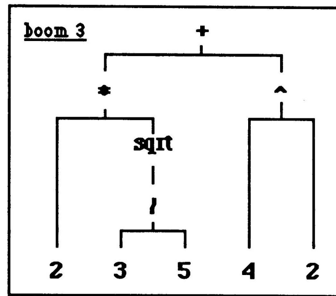
2 * sqrt (3/5 + 4^2)

met boom 2. Functoren gaan voor operatoren. Bijvoorbeeld de term

2 * sqrt(3/5) + 4^2

waarin **sqrt(3/5)** genoteerd is als een functor met een argument, heeft de structuur van boom 3.





Voor dit voorbeeld had **Sqrt** dus niet als operator gedeclareerd hoeven te zijn. Operatoren geven wat mee mogelijkheden dan predikaten wat het gebruik van spaties betreft, zoals blijkt uit het volgende voorbeeld:

2 * sqrt (3/5) + 4^2

Deze term, die van de vorige slechts een spatie verschilt, is syntactisch alleen acceptabel als **sqrt** een operator is. Operatoren mogen niet tegen elkaar aan staan:

correct:	**not not P**	**2* -(3+4)**	
fout:	**notnot P**	**2*-(3+4)**	/* FOUT ! */

In het algemeen is het verstandig om haakjes te gebruiken bij twijfel over de interpratie van termen. Zo wordt ons eerste, met boom 1 corresponderende voorbeeld duidelijker als we noteren:

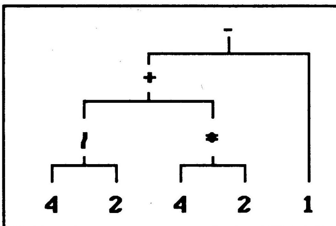
2 * (sqrt 3) / 5 + 4^2

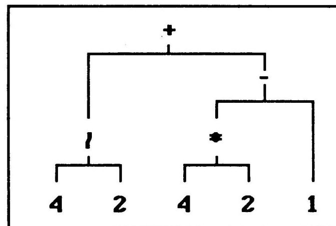
10.3.3 Associativiteit

Als we geen haakjes gebruiken, kan het gebeuren dat een operator concurreert met een operator van gelijke priorititeit. Bijvoorbeeld het plus- en het minteken in:

4/2 + 4*2 - 1.

De twee andere operatoren (/ en *) geven geen probleem, die gaan op grond van hun *precedence in ieder geval* voor. Er zijn twee analyses mogelijk: eerst optellen (linker boompje) of eerst aftrekken (rechter boompje):





Dergelijke volgordeproblemen zijn in Prolog opgelost door aan elke operator behalve een prioriteit ook een *associativiteit* toe te kennen. De *associativiteit* geeft aan welke van twee operatoren voorgaat de linker of de rechter. Er zijn drie mogelijkheden:

(1) *linksassocitatief*: de linker operator gaat voor;
(2) *rechtsassociatief*: de rechter operator gaat voor
(3) *niet-associatief*: de operator mag niet met zichzelf concurreren

*****, **/**, **+** en **-** zijn voorbeelden van operatoren die *linksassociatief* zijn, **^** en **sqrt**, maar ook conjunctie (**,**) en disjunctie (**;**), zijn voorbeelden van *rechtsassociatieve* operatoren.

Aangezien + en - linksassociatief zijn correspondeert de expressie in ons voorbeeld met het linker boompje hierboven. Zonder operatoren geschreven is de term dus:

```
-(+(/(4,2),*(4,2)),1).
```

Een voorbeeld van een *niet-associatieve operator* is de operator is:

```
Var is 4/2 + 4*2 - 1
```

wordt door de interpretator geanalyseerd als:

```
is(Var,-(+(/(4,2),*(4,2)),1))
```

Het niet-associatief zijn van **is** heeft tot doel onzinnige expressies snel af te straffen, zoals hieronder:

```
| ?- Var1 is Var2 is 4/2 + 4*2 - 1.
Syntax error
```

De foutmelding is het gevolg van het niet-associatief zijn van **is**. De *syntactische* problemen kunnen opgelost worden met *haakjes*, maar dan volgt een *semantische* afstraffing:

```
| ?- Var1 is (Var2 is 4/2 + 4*2 - 1).
Illegal arithmetic expression
```

Een ander voorbeeld van een niet-associatieve operator is het indien-symbool (**:-**).
Dit geldt zowel voor de binaire als de unaire versie. Zoals blijkt uit het voorbeeld, dat besproken werd in 9.11, hoeft het gebruik van haakjes om het niet-associatief zijn te omzeilen, bij déze operator niet tot semantische problemen te leiden.

10.4 op/3 en current_op/3

Om een functor of predikaat als operator te kunnen gebruiken moet de interpretator beschikken over de *prioriteit*, het aantal *argumenten*, de plaats en de *associativiteit* van de operator. De laatste drie gegevens kunnen gecombineerd worden weergegeven door gebruik te maken van speciale symbolen, opgebouwd uit de letters ***f***, ***x*** en ***y***, die staan voor:

f de te declareren functor;

x een argument dat óf geen operatoren bevat, óf alleen operatoren die voorafgaan aan *f*, dus met een *prioriteitsgetal* lager dan dat van *f*;

y een argument dat óf geen operatoren bevat, óf operatoren van gelijke prioriteit als *f*, óf operatoren die voorafgaan aan *f*.

De volgende typen operatoren zijn mogelijk:

type	*arg*	*positie*	*associativiteit*
yfx	2	infix	linksassociatief
xfy	2	infix	rechtsassociatief
xfx	2	infix	niet associatief
fy	1	prefix	rechtsassociatief
fx	1	prefix	niet associatief
yf	1	postfix	linksassociatief
xf	1	postfix	niet associatief

Het lastigste van deze typeaanduiding is de associativiteit. Een ezelsbruggetje is: de plaats van *y* geeft de associativiteit aan. (Bevat de typeaanduiding geen *y*, dan ..., inderdaad!) Bovenstaande typeaanduidingen worden in het **op**/3 predikaat gebruikt om een operator te declareren (aangezien operatordeclaraties meestal als *directief* in een file voorkomen gebruiken we die vorm in onderstaande omschrijving, in de hoop daarmee pogingen het **op** predikaat te herdefiniëren te voorkomen, zie 2.8.4):

:-op(***Prio***,***Type***,***Oper***). declareert ***Oper*** als een operator van het type ***Type*** met een prioriteit gelijk aan ***Prio***. Alle argumenten moeten gebonden zijn.

:-op(0,***Type***,***Oper***). heft de declaratie van ***Oper*** als operator van het type ***Type*** op (prioriteit=0).

Tussen welke grenzen de prioriteit ligt, hangt af van de Prologinterpretator. In oudere interpretatoren ligt de prioriteit meestal tussen 0 en 255 (CPROLOG bijvoorbeeld). In de nieuwere interpretatoren ligt de prioriteit tussen 0 en 1200. Het *indien-teken* heeft altijd de hoogst mogelijke waarde voor de prioriteit. Informatie over operatoren kan verkregen worden met **current_op**/3:

current_op(Prio,Type,Oper) unificeert ***Prio*** met de prioriteit, ***Type*** met het type van de operator ***Oper***. Elk van drie argumenten mag een variabele zijn.

Bijvoorbeeld:

```
| ?-current_op(P,T,:-).
P=1200,
T=fx;                          /* unair, prefix */
P=1200,                        /* Nog een oplossing */
T=xfx                          /* binair, niet-associatief */
yes
```

Een ander voorbeeld:

```
| ?-current_op(P,T,:=>).
no                             /* :=> is geen operator  */
```

```
| ?-op(700,xfx,:=>).                    /* :=> invoeren als operator */
yes
| ?-current_op(P,T,:=>).
P=700
T=xfx
yes                                     /* :=> inderdaad gedeclareerd */
```

Nu is de volgende vraag *syntactisch* in orde:

```
| ?-geef_kenmerk(getal:=>G,[persoon:=>1,getal:=>enkelvoud]).
no
```

De operatordeclaratie van :=> kan opgeheven worden door middel van:

```
| ?-op(0,xfx,:=>).                      /* :=> opgeheven */
yes
```

De volgende vraag is daarmee uiteraard syntactisch onaanvaardbaar geworden:

```
| ?-zoek(getal:=>G,[persoon:=>1,getal:=>enkelvoud]).
Syntax error
```

10.4.1 De namen van operatoren

De naam van een operator kan elk teken bevatten, maar mag niet beginnen met een hoofdletter of met een laagliggend streepje. Als het nodig is om de operator in het **op** predikaat tussen apostrophen te zetten, moeten die apostrophen ook bij gebruik van de operator geplaatst worden. De naam is dan niet handig gekozen. Voor de meeste operatoren wordt een atoom bestaande uit niet-alfabetische tekens gebruikt. Maar door goedgekozen woorden als operator te gebruiken, kunnen clausules soms op gewoon Nederlands gaan lijken (zie opgave *10.5*).

10.4.2 Herdeclaratie van een operator

Als een operator opnieuw gedeclareerd wordt, zal de interpretator die declaraties welke met de nieuwe declaratie onverenigbaar zijn, opheffen. Het is zelden zinvol een operator meer dan één keer te declareren; + en - die zowel unair als binair gedeclareerd zijn, zijn uitzonderingen.

Met operatoren moet niet te veel worden gestrooid. In het Engels wordt de mogelijkheid om operatoren te declareren vaak omschreven als *syntactic sugar*. Het is de kunst te voorkomen dat een programma er door onhandig gekozen operatoren "besuikerd" uit gaat zien.

Na deze behandeling van operatoren zijn alle syntactische verschijnselen verklaard, die zich in Prolog voor kunnen doen. Alhoewel ietwat versluierd door operatoren, de lijstnotatie en de voor I/O operaties vereiste punten en *white space*, kent Prolog slechts één syntactische eenheid en dat is de term die bestaat uit een functor met argumenten.

10.5 Opgaven

10.1 Herdefinieer het predikaat **geef_kenmerk**/2 uit de tekst, zodanig dat **:=>** als scheidingsteken gebruikt mag worden in de lijst van kenmerken.

10.2 Onderzoek of **not** een operator is; is dat niet het geval, declareer **not** dan als operator en definieer **not** zonodig ook als predikaat.

10.3 Schrijf een predikaat **alle_operatoren** dat alle operatoren met hun kenmerken naar het scherm schrijft:

```
| ?-alle_operatoren.

?- fx 1200
:- fx 1200
--> xfx 1200
:- xfx 1200
; xfy 1100
```

(*enzovoorts*).

Uit het door **alle_operatoren** geproduceerde overzicht (een mooiere layout is toegestaan!) valt o.a. af te leiden wat in een bepaalde Prologimplementatie de hoogst mogelijke waarde voor de prioriteit is. Ook kunnen we dit overzicht gebruiken om onze eigen operatoren op het juiste prioriteitsniveau in te voegen. (Het verkorte overzicht hierboven begint overigens met een niet besproken operator, namelijk: **?-** , die blijkens het lijstje een unaire prefix operator is. In sommige Prologimplementaties is dit een alternatief symbool voor de unaire operator **:-** waarmee directieven aangegeven worden. De prompt heeft niet voor niets deze vorm, want de betekenis van **?-** en **:-** is immers te omschrijven als: *onderzoek de volgende formule op afleidbaarheid.*)

10.4 Declareer de operatoren die nodig zijn om de juiste structuur te geven aan de rekenexpressies uit de tekst (10.3) en aan de volgende formule:

2 * sqrt sqrt 1024 kwadraat + 2^3^4

De normale declaraties van de systeemoperatoren * en + mogen hierbij niet gewijzigd worden. **kwadraat** moet zo gedeclareerd worden dat

2 kwadraat kwadraat

syntactisch niet geaccepteerd wordt (*syntax error*). Gebruik **display** om de structuur van een term te controleren:

```
| ?-display(2 * sqrt sqrt 1024 kwadraat + 2^3^4).
+(*(2,sqrt(sqrt(kwadraat(1024)))),^(2,^(3,4)))
yes
```

10.5 Declareer de operatoren die nodig zijn om de clausules op de volgende baldzijde syntactisch acceptabel te maken (sommige zijn *unair*).
Afgezien van de syntactische aspecten, is dit voorbeeld interessant doordat klassen en individuen gedefinieerd zijn, waarbij een individu kenmerken ontleent aan de klasse waartoe dat individu behoort. Dit wordt ook wel als volgt uitgedrukt: tussen klassen en individuen bestaat een *overervingsrelatie*. Ook tussen verschillende klassen kan overerving van kenmerken plaats vinden.
Een relatie tussen een *individu* en een *klasse* wordt beschreven met de relatie **is_een**. Bijvoorbeeld : **oeioei is_een twatwa.**
Relaties tussen *klassen* worden beschreven met **zijn.** Een voorbeeld daarvan is: **twatwas zijn vogels.**

```
lala is_een twatwa.
oeioei is_een twatwa.
I is_een Y :-
        nonvar(Y),
        I is_een X,
        meervoud(X,Xs),
        Xs zijn Ys.

twatwas zijn vogels.
dieren zijn levende_wezens.
Xs zijn Zs :-
        nonvar(Zs),
        Xs zijn Ys,
        Ys zijn Zs.

vogels hebben veren.
vogels hebben 2 vleugels.
vogels hebben 2 poten.
dieren hebben ademhaling.
dieren hebben hersens.
Xs hebben E :-
        Xs zijn Ys,
        Ys hebben E.

X heeft E :-
        X is_een Y,
        meervoud(Y,Ys),
        Ys hebben E.
```

Zoals uit de sessie hieronder blijkt, bezit **oeioei** eigenschappen van **vogels**, die niet expliciet als eigenschappen van **oeioei** genoemd werden, bijvoorbeeld: **oeioei heeft 2 poten**.

```
| ?-oeioei is_een X.
X=twatwa;                       /* ';' geeft niet nog een oplossing */
no
| ?-oeioei is_een dier.
yes                             /* maar verificatie kan wel */
| ?-oeioei heeft 2 poten.
yes
| ?-dieren hebben E.
E=ademhaling,
E=hersens,
no
| ?-twatwas hebben E.
E=veren;                        /* Eigenschap van vogels */
E=2 vleugels;                   /* Eigenschap van vogels */
E=2 poten;                      /* Eigenschap van vogels */
E=ademhaling;                   /* Eigenschap van dieren */
E=hersens;                      /* Eigenschap van dieren */
no
```

Voeg aan het voorbeeld de ontbrekende clausules toe, zodat het verschijnsel van *overerving van*

eigenschappen gedemonstreerd kan worden met de sessie hierboven. Breid vervolgens het aantal objecten en eigenschappen uit.

10.6 In opgave *5.8* hebben we in de file *match.pro* het predikaat **match/2** gedefinieerd. We breiden de mogelijkheden om patronen op te geven uit, zodat datgene waarmee een ?, + of * matcht geünificeerd wordt met een variabele:

Symbool	*Correspondeert met:*	*Unificatie*
?	1 element	
?Provar	1 element (el)	**Provar =** ***el***
*	0 of meer elementen	
***Provar**	0 of meer element (el1, el2, ...)	**Provar=[*el1,el2,...*]**
+	1 of meer elementen	
+Provar	1 of meer elementen (el1, el2, ...)	**Provar=[*el1,el2,...*]**

Voorbeelden van gebruik van de nieuwe mogelijkheden van match zijn:

```
| ?-match([kleur,?V,?K],[kleur,appel,rood]).
V = appel,
K = rood
yes
| ?-match([x,*VAR,y,+VAR,z],[x,a,b,c,y,a,b,c,z]).
VAR = [a,b,c]
yes
```

Een groot gedeelte van de nieuwe definitie van **match** staat hieronder:

```
match([],[]):-!.

match([?|Tp],[_|Tl]):-
        match(Tp,Tl).
match([?E|Tp],[E|Tl]):-                /* Nieuwe clausule */
        match(Tp,Tl).

match([*|Tp],[_|Tl]):-
        match(Tp,Tl).
match([*|Tp],[_|Tl]):-
        match([*|Tp],Tl).
match([*[H]|Tp],[H|Tl]):-              /* Nieuwe clausule */
        match(Tp,Tl).
match([*[H|T]|Tp],[H|Tl]):-            /* Nieuwe clausule */
        match([*T|Tp],Tl).

match([X|Tp],[X|Tl]):-
        match(Tp,Tl).
```

(a) Declareer de vereiste operatoren.
(b) Implementeer de patroonsymbolen + en +***Variabele.***
(c) Hofstadter (1979) heeft het **miu**-systeem bedacht. Het systeem beschrijft symboolreeksen die opgebouwd zijn uit de letters **m**, **i** en **u**. Uit een symboolreeks kunnen andere reeksen ontstaan door toepassing van de volgende vier *herschrijfregels* (- en *x* stellen willekeurige reeksen voor):

(1) -i => **-iu** Aan een **i** op het einde van een reeks mag een **u** worden toegevoegd.
(2) **m***x* => **m***xx* Begint een reeks met **m**, dan mag de reeks daarachter verdubbeld worden.
(3) **-iii-** => **-u-** 3 **i**'s mogen vervangen worden door een **u**.
(4) **-uu-** => **--** 2 **u**'s mogen weggelaten worden.

Er is een *startreeeks* **mi**.
We kunnen vanuit de startreeks generaties van nieuwe symbolen produceren, waarbij generatie *n* bestaat uit de symbolen die ontstaan na *n* keer een regel toegepast te hebben (de toegepaste regel staat tussen haakjes):

Generatie 0:
  mi.
Generatie 1:
  miu(1), **mii**(2).
Generatie 2:
  miu kan herschreven worden tot: **miuiu**(1).
  mii kan herschreven worden tot: **miiu**(1), **miiii**(2).
Generatie 3:
  miuiu geeft: **miuiu**(2),
  miiu geeft: **miiuiiu**(2),
  miiii geeft: **miiiiu**(1), **miiiiiiii**(2), **mui**(3), **miu**(3).
enzovoorts.

Gebruik **match/2** om elk van de vier herschrijfregels in Prolog te implementeren (representeer de symboolreeksen als lijsten).
(d) Schrijf vervolgens een predikaat **mu/1** dat uit de beginreeks **mi** nieuwe reeksen berekent en wel *breadth-first*. Dit houdt in dat de symboolreeksen geproduceerd worden op de bovenbeschreven wijze: generatie voor generatie.(Tip: Gebruik **assertz** en **retract** om tussenresultaten te bewaren en op te ruimen.) (Het is Hofstadter niet gelukt de reeks **mu** met behulp van de vier regels uit de beginreeks **mi** af te leiden. Zou het met dit programma wel lukken?)

10.7 In hoofdstuk 7 werden de predikaten **gpg** en **gxg** gedefinieerd, die twee als lijsten gespecificeerde getallen respectievelijk optellen en vermenigvuldigen en het resultaat in een derde variabele zetten. We introduceren nu een binaire operator voor "lijstgetallen", vergelijkbaar met **is** voor gewone getallen. We kiezen voor deze operator het symbool **<-** en we voegen het volgende directief toe aan de file *reken.pro* met de rekenpredikaten:

```
:-op(700,xfx,<-).
```

Deze operator maakt ondermeer de volgende expressies voor de interpretator syntactisch aanvaardbaar:

```
Var <- E1 + E2
Var <- E1 * E2
Var <- E1 + E2 + E3
Var <- E1 * E2 * E3
Var <- (E1 + E2) * E3
```

Maar zoals vaker gezegd, syntactische aanvaardbaarheid betekent nog niet dat er iets gebeurt:

```
| ?-reconsult('calcula.pro').            /* definieert: gpg en gxg */
yes
```

```
| ?-Var <- [1,0]+[1,0].
no
| ?-Var <- [1,0]*[1,0].
no
```

Voor eenvoudige berekeningen als hierboven zijn de volgende definities voldoende:

```
Var <- E1 + E2 :- gpg(E1,E2,Var).
Var <- E1 * E2 :- gxg(E1,E2,Var).
```

Nu gebeurt er ineens wel wat (we nemen aan dat ook bovenstaande definities zich in de file *calcula.pro* bevinden).

```
| ?-reconsult('calcula.pro').
yes
| ?-Var <- [1,0]+[1,0].
Var=[2,0]
yes
| ?-Var <- [1,0]*[1,0].
Var=[1,0,0]
yes
```

We hadden in plaats van **gpg** en **gxg** te definiëren *met behulp van* **<-** de hoofden in de definities van **gpg** en **gxg** textueel kunnen veranderen in termen met een pijl; **gpg** en **gxg** zouden dan ophouden te bestaan. Dit zou bij het gebruik van **<-** een inferentiestap schelen. Wat nu nog niet werkt zijn expressies waarin de operator meer dan één keer voorkomt, al dan niet met haakjes, zoals:

```
Var <- E1 + E2 + E3
Var <- E1 * E2 * E3
Var <- E1 * (E2 * E3)
Var <- (E1 * E2) * E3
```

Deze gevallen gaan fout, omdat **gpg** en **gxg** *samengestelde* formules te verwerken krijgen, bijvoorbeeld:

```
Var <- E1 + E2 + E3
```

is vanwege het linksassociatief zijn van + equivalent met:

```
Var <- (E1 + E2) + E3
```

en dit leidt tot een toepassing van **gpg** die er als volgt uitziet:

```
gpg(E1+E2,E3,Var)
```

Een formule moet recursief eerst helemaal worden afgebroken voordat **gpg** aan het werk wordt gezet.

Breid nu de definitie voor **<-** stapsgewijs uit, zodanig dat expressies met een willekeurige combinatie van + en *, al dan niet met haakjes, correct worden berekend.

Uiteindelijk moet een sessie als de volgende mogelijk zijn:

```
| ?-V  <-  [1,2,3,4,5,6].
V=[1,2,3,4,5,6]
yes

| ?-Goedelgetal_van_0_is_0  <-                /* zie Nagel& Newman(1958) */
|     [2]*[2]*[2]*[2]*[2]*[2]   *
|     [3]*[3]*[3]*[3]*[3]   *
|     [5]*[5]*[5]*[5]*[5]*[5].
Goedelgetal_van_0_is_0=[2,4,3,0,0,0,0,0,0]
yes

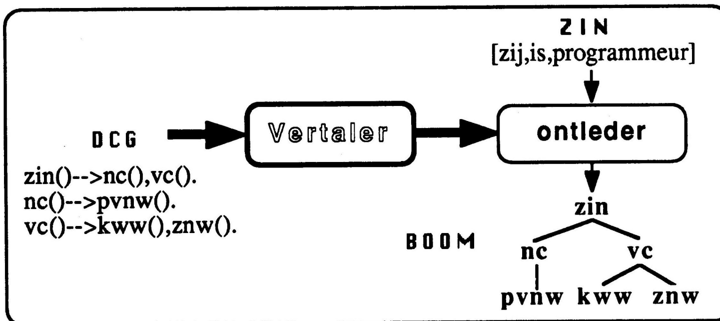
| ?-V <- [1,2] * [1,0,0] + [3,4].
V=[1,2,3,4]
yes

| ?-V  <-  ([1,1]+[1])*[1,0]*[1,0]+[3,4].
V=[1,2,3,4]
yes

| ?-V  <-  [2]*[3,2]+[4,5]*[1,0]+[6]*[1,0,0,0,0].
V=[6,0,5,1,4]
yes
```

11 Logische Grammatica's

Een grammatica is een verzameling regels volgens welke complexe gehelen uit elementen kunnen worden opgebouwd. Er zijn onder meer grammatica's op te stellen voor logische formules, voor programmeertalen, voor geometrische figuren, voor scheikundige formules, voor kunstmatige talen als Esperanto en voor verzamelingen zinnen uit een natuurlijke taal. In dit hoofdstuk wordt het paradigma geschetst dat door gebruikers van logische programmeer - talen gevolgd wordt bij hun pogingen vorderingen op grammaticaal terrein te maken.
Een computerprogramma dat kan onderzoeken *of* een reeks van elementen volgens de regels van een bepaalde grammatica is opgebouwd, heet een *acceptor* voor die grammatica; een acceptor die (bijvoorbeeld door middel van een boomstructuur) aangeeft *hoe* de regels toegepast werden om een bepaald object te construeren heet, een *ontleder* of *parser*. Een programma dat objecten *voortbrengt* volgens de regels van een grammatica, heet een *generator*. Een *logische grammatica* is een grammatica waarvan elke regel vertaald kan worden in een clausule van een logische programmeertaal; de aldus ontstane clausules vormen tezamen een ontleder of generator in die programmeertaal. Alhoewel er druk geëxperimenteerd wordt met de ontwikkeling van parsers in parallel werkende logische programmeertalen (zie bijvoorbeeld Goos en Harmanis, 1986), is in de huidige praktijk Prolog de programmeertaal die voor het ontwikkelen van logische grammatica's het meest gebruikt wordt. Prolog is daarbij zowel doeltaal (de taal waarnaar de parser/generator vertaald wordt) als implementatietaal (de taal waarin de vertaler geschreven is). Prototypisch voor de werkwijze in Prolog is het *definite clause grammar* (DCG) formalisme. Volgens dit formalisme bestaat een grammatica (een DCG) uit een aantal DCG-regels. Elke regel wordt vertaald naar een Prologclausule. We kunnen (onder meer) DCG's opstellen voor het ontleden en voor het genereren van zinnen. De zinnen worden hierbij gerepresenteerd als lijsten van woorden. In eenvoudige gevallen is de acceptor of ontleder voor een DCG tevens generator voor die DCG. Wij zullen ons voornamelijk met zinsontleding bezighouden. Hoe grammatica, regelvertaler, ontleder, zin en geproduceerde structuur daarbij samenhangen, is hieronder schematisch weergegeven:



In Prologsystemen die het DCG formalisme ondersteunen worden alle DCG-regels in een file die ge(re)**consult** wordt, automatisch vertaald naar normale Prologclausules. De DCG bevindt zich dus alleen in vertaalde vorm in de gegevensbank.

11.1 Definite Clause Grammars: het formalisme

Zoals gezegd bestaat een DCG uit een aantal regels. Een DCG-regel lijkt erg op een gewone clausule (en is daarom ook vrij eenvoudig daarin om te zetten). Een regel is opgebouwd uit *operatoren*, *nonterminals*, *terminals* en *condities*.

11.1.1 De operatoren

De hoofdoperator in een DCG-regel is de *herschrijfpijl* (-->), een binaire, niet-associatieve operator. Een DCG-regel kan gelezen worden als: wat links van de pijl staat "*kan herschreven worden tot*" of "*kan bestaan uit*" datgene wat rechts van de pijl staat. De andere operatoren die gebruikt kunnen worden zijn de gewone clausulebouwers: *komma* (voor *conjunctie*) en *puntkomma* (voor *disjunctie*). DCG-regels zijn als volgt samengesteld:

> *links* van de pijl mag een *nonterminal* staan.
>
> *rechts* van de pijl mogen door komma's en puntkomma's van elkaar gescheiden *nonterminals*, *terminals* en *condities* staan. Waar nodig mogen *haakjes* gebruikt worden om de volgorde van verwerking te beïnvloeden.

Een DCG-regel moet worden afgesloten door een punt plus *white space*. Een voorbeeld van een DCG-regel is:

```
zin-->nc, vc.
```

Deze regel is te lezen als "een *zin kan bestaan uit* een *nc* gevolgd door een *vc*".

11.1.2 Nonterminals

Een *nonterminal* is een benoemd deel van een zin (eventueel de gehele zin). De mogelijke structuren van het door de *nonterminal* bestreken zinsdeel (moeten) worden beschreven door herschrijfregels. (DCG-vertalers controleren niet of een nonterminal wel ergens in de grammatica een keer links van de pijl voorkomt, dit wordt aan de grammaticus toevertrouwd.)
In DCG-regels worden *nonterminals* geschreven als gewone proposities. Argumenten zijn toegestaan en, zoals we verderop zullen zien, vaak onontbeerlijk.
In 11.1.1 zijn al als *nonterminal* opgetreden: **zin**, **nc**, en **vc**. Ingewikkelder versies van deze nonterminals, alle met twee argumenten, zien we in de volgende DCG-regel:

```
zin([],zin(nc,vc))-->nc([P,G],nc),    vc([P,G],vc).
```

11.1.3 Terminals

Een *terminal* is een element dat voor kan komen in een *zin*, die behoort tot de verzameling zinnen die door een DCG beschreven wordt. In een DCG-regel staan de *terminals* in een lijst en wordt de lijst zelf ook *terminal* genoemd. In het DCG-formalisme wordt de te analyseren zin gerepresenteerd als lijst. Een terminal in een DCG laat als het ware een stukje van zo'n inputlijst zien. Bijvoorbeeld:

```
idioom-->[de, appel], vallen, [niet, ver, van, de, boom], [!].
vallen-->[viel];[valt].
```

Merk op dat [!] een terminal is en níet het *cut*predikaat. De nonterminal **vallen** kan tot twee verschillende terminals herschreven worden, waardoor de drie regels hierboven zowel de tegenwoordige als de verleden tijd van de idiomatische uitdrukking beschrijven. (In de DCG-regel in 11.1.2 kwamen weliswaar lijsten voor, maar dat waren *argumenten* van nonterminals en géén terminals!)

11.1.4 Condities

De laatste bouwsteen, de *conditie,* kan alleen aan de rechterkant van de herschrijfpijl voorkomen. Een *conditie* bestaat uit een aantal doelproposities tussen *accolades*, gescheiden door komma's of puntkomma's. Wat tussen de accolades staat vinden we letterlijk terug in de vertaling van de regel. Met behulp van accolades kunnen we dus stukjes Prolog aan onze regels toevoegen. Vaak wordt deze faciliteit gebruikt om extra condities aan een DCG-regel toe te voegen, waardoor de regel alleen toepasbaar is als aan die condities voldaan wordt. Ook worden condities toegepast ter ondersteuning van het bouwen van structuren tijdens het ontleedproces. Een bijzondere "conditie" is het *uitroepteken.* Deze hoeft niet tussen accolades te worden gezet. Het niet altijd te vermijden gebruik van het uitroepteken in een DCG is er vaak de oorzaak van dat een DCG slechts in één richting werkt (óf als generator óf als ontleder). Een andere oorzaak van onomkeerbaarheid is het gebruik van onomkeerbare normale condities.
We hadden de nonterminal **idioom** uit het vorige voorbeeld ook met behulp van een *conditie* kunnen definiëren:

```
idioom-->
        [de, appel],
        [V], {is_elem_van(V,[viel,valt])},
        [niet, ver, van, de, boom],
        [!].
```

Hierin staat [V] voor een *variabele terminal*: *V* wordt geünificeerd met het eerste woord dat volgt op *appel*. Na de unificatie wordt in een conditie door middel van het predikaat **is_elem_van** gecontroleerd of *V* één van de twee toegestane werkwoorden is. Aangezien **is_elem_van** in twee richtingen werkt, kunnen we met **idioom** ook (twee) zinnen *genereren* (wat overigens ook met de vorige versie mogelijk is).

11.2 Acceptor 1 (geen argumenten, geen condities)

Aan de hand van een voorbeeldgrammatica die we stap voor stap uitbreiden, zullen we zien wat we met de verschillende bouwstenen van het DCG formalisme kunnen doen.

De eerste versie van onze grammatica is als volgt:

```
zin-->nc, vc.
nc-->pvnw.
nc-->znw.
vc-->kww, nc.
pvnw-->[zij].
kww-->[is].
znw-->[programmeur].
```

Deze regels worden door de regelvertaler van Prolog omgezet in het volgende programma:

```
zin(A,B):-
      nc(A,C),
      vc(C,B).
nc(A,B):-
      pvnw(A,B).
nc(A,B):-
      znw(A,B).
vc(A,B):-
      kww(A,C),
      nc(C,B).
pvnw([zij|B],B).
kww([is|B],B).
znw([programmeur|B],B).
```

Over de vertaling valt het volgende op te merken:

—We zien dat elke DCG-regel vertaald wordt in een *clausule.*
—Elke *nonterminal* wordt vertaald in een propositie met *twee* argumenten.
—*Terminals* vinden we terug als element van een lijst in een argument van een propositie. (Bijvoorbeeld de terminal *programmeur,* is de kop van de lijst in het eerste argument is van de propositie **znw([programmeur|B],B)**.)

Na vertaling (en toevoeging aan de gegevensbank) is er voor iedere nonterminal een predikaat gedefinieerd. Met deze predikaten kunnen we onderzoeken of een of meer elementen aan het *begin* van een lijst van woorden tezamen een bepaalde structuur bezitten. Hiertoe moeten we een met die structuur corresponderende nonterminal met *twee argumenten* als doelpropositie aanbieden waarbij

het eerste argument de *te onderzoeken lijst* moet bevatten en
het tweede bestemd is voor de overblijvende, *niet tot de structuur behorende elementen.*

Willen we bijvoorbeeld weten of de lijst *[zij,is,programmeur]* met een naamwoord - constituent (*nc*) begint, dan gebruiken we het predikaat **nc**/2:

```
| ?-nc([zij,is,programmeur],Rest).
Rest=[is,programmeur]
yes
| ?-nc([zij,khomeiny,beledigd],Rest).
Rest=[khomeiny,beledigd]
yes
```

Voorbeelden van niet-acceptatie zien we hieronder:

```
| ?-nc([zij,is,programmeur],[]).        /* De lijst in zijn geheel is geen nc */
no
| ?-nc([is,zij,programmeur],Rest).      /* De lijst begint met een koppelww */
no
```

Willen we weten of de woorden in een lijst een door de grammatica beschreven *zin* vormen, dan gebruiken we de tot predikaat verheven nonterminal **zin**/2:

```
| ?-zin([zij,is,programmeur],[]).
yes
```

```
| ?-zin([zij,is,programmeur],Rest).
Rest=[]
yes
| ?-zin([is,zij,programmeur],[]).
no
```

11.2.1 De woordacceptoren

Om te begrijpen hoe de grammaticaclausules werken, bekijken we allereerst de werking van de regels die het laagst in de grammatica staan, de drie *woordacceptoren*:

```
pvnw([zij|B],B).
kww([is|B],B).
znw([programmeur|B],B).
```

We zien dat het eerste argument van een lijst gelijk moet zijn aan een bepaald woord om unificatie en dus acceptatie mogelijk te maken. Het tweede argument (***B***) wordt geünificeerd met de staart, bijvoorbeeld:

```
| ?-pvnw([zij,is,programmeur],Rest).
Rest=[is,programmeur]
yes
| ?-pvnw([zij],Rest).
Rest=[]
yes
```

De woordacceptoren kunnen ook (woorden) genereren:

```
| ?-pvnw([Woord],[]).
Woord=zij;
no
```

We hebben slechts één persoonlijk voornaamwoord in onze grammatica opgenomen, waardoor we ook maar één oplossing krijgen.

11.2.2 De constituentacceptoren

11.2.2.1 Enkelvoudige constituenten

De woordacceptoren worden gebrukt om de acceptoren van constituenten te definiëren. Zo bestaat volgens onze grammatica een naamwoordconstituent (**nc**) uit een persoonlijk voornaamwoord (**pvnw**) of uit een zelfstandig naamwoord (**znw**):

```
nc-->pvnw.
nc-->znw.
```

In de vertalingen voor deze regels zien we woordacceptoren optreden, waarbij de argumenten van een **nc** (***A*** en ***B***) gelijk zijn aan die van de gebruikte woordacceptor:

```
nc(A,B) :-
    pvnw(A,B).
nc(A,B) :-
    znw(A,B).
```

11.2.2.2 Conjuncties van constituenten

In de meeste gevallen is een constituent samengesteld. Als voorbeeld bekijken we de vertaling van de regel die de verbale constituent beschrijft:

```
vc-->kww, nc.
```

De vertaling hiervoor luidt:

```
vc(A,B) :-
        kww(A,C),
        nc(C,B).
```

De te accepteren woordreeks bevindt zich in de variabele *A*. Deze treedt ook op als eerste argument in de koppelwerkwoordacceptor. Blijkt het eerste element van de woordreeks een koppelwerkwoord te zijn, dan wordt de lijst van overblijvende elementen geünificeerd met de *tussenvariabele C*. Deze variabele treedt ook op in nc/2. Vormen de beginelementen van *C* een *nc*, dan wordt wat van de lijst resteert geünificeerd met variabele *B* die ook het tweede argument in het hoofd van de regel is. Een dergelijke *keten van variabelen* zien we ook bij meer dan twee conjuncten: *elke acceptor verwerkt een stukje van de te accepteren lijst en geeft de rest door aan de volgende acceptor.* Wat er na de conjunctie van acceptoren van de lijst overblijft, wordt geünificeerd met de restvariabele in het hoofd van de clausule.

Hiermee hebben we alles besproken wat van belang is voor het begrijpen van de werking van de eerste versie van onze DCG: **zin/2** zoekt eerst naar beginelementen die samen een *nc* vormen; de elementen die overblijven moeten een *vc* vormen.

11.2.3 Acceptor 1 als generator

Omdat onze woordacceptor in twee richtingen werkt kunnen we ook (alle vier mogelijke) zinnen genereren:

```
| ?-zin(Zin,[]).
Zin    = [zij,is,zij];
Zin    = [zij,is,programmeur];
Zin    = [programmeur,is,zij];
Zin    = [programmeur,is,programmeur];
no
```

We kunnen ook alvast een woord invullen, zoals hieronder, zodat alleen de zinnen die beginnen met *zij* geproduceerd worden:

```
| ?- nl,
    zin([zij,W1,W2],[]),
    write(zij),put(32),write(W1),put(32),write(W2),nl,
    fail.

zij is zij
zij is programmeur
no
```

11.2.4 Opgaven

11.1 (a) Zet de grammatica uit de tekst in de file *grammal.pro*. Consulteer deze file en experimenteer met de vertaalde grammatica. Gebruik **listing**/1 om de vertaling van de DCG te bestuderen. In veel gevallen zal blijken dat de variabelen van de interpretator andere namen hebben gekregen dan in de tekst staat aangegeven.
(b) Definieer de constituent ***nc*** met behulp van disjunctie (alleen bij wijze van oefening, niet omdat dat hier bijzonder handig is). Bekijk de vertaling van de nieuwe definitie voor **nc**/2.
(c) Voeg enige woorddefinities toe aan de grammatica (zie het begin van de volgende sectie).

11.3 Acceptor 2 (argumenten)

De eerste versie van onze grammatica had een zeer beperkt woordenboek. We kunnen een heleboel woorden toevoegen zonder dat er foute zinnen door de grammatica beschreven worden (*linguïst, filosoof, hij,* etcetera). Problemen ontstaan wanneer we de woorden *ik* en *ben* toevoegen met behulp van de volgende regels:

```
pvnw-->[ik].
kww-->[ben].
```

Nu worden weliswaar een aantal nieuwe correcte zinnen geaccepteerd en gegenereerd, zoals

```
[ik,ben,ik];
[ik,ben,programmeur];
```

maar ook een heleboel, waarin iets mankeert aan de *correspondentie* tussen het *onderwerp* en de *persoonsvorm* (*subject-verb agreement*). Voorbeelden daarvan zijn:

```
[ik,is,programmeur];
[zij,ben,programmeur];
[programmeur,is,ik];
```

Wat we nodig hebben zijn *syntactische kenmerken* (zoals we die ook al in hoofdstuk 3 en 4 gezien hebben).

Om kenmerken in onze DCG in te voeren, maken we gebruik van *nonterminals met argumenten*. De truc is om in een regel alle nonterminals waarvan de *kenmerken* gelijk moeten zijn aan elkaar uit te rusten met *argumenten* die gelijk zijn aan elkaar. In de tweede versie van onze minigrammatica is dat gebeurd. Hierbij hebben we als niet noodzakelijke bijzonderheid de kenmerken bij elkaar in een lijstje gezet (zoals ook in opgave 4.6 het geval was), waardoor elke nonterminal *in de grammatica* slechts één argument bezit (en niet voor elk kenmerk één).
Ons congruentieprobleem is opgelost door in de eerste regel **nc** en **vc** dezelfde kenmerkenlijst *[P,G]* (persoon en getal) te geven. Is de *nc* een persoonlijk voornaamwoord, dan komen de ***G*** en de ***P*** uiteindelijk uit een *lexicale regel* (zo dopen we de regels die eindigen in een *terminal*; alle lexicale regels bij elkaar vormen het *lexicon* (*woordenboek*)). De zelfstandige naamwoorden bevatten als enig kenmerk *getal*, bijvoorbeeld *enkelvoud*. Voor de congruentie tussen een *nc* en een *vc* in een *zin* is behalve het getal (***G***) ook de persoon (***P***) nodig. Een naamwoordconstituent (*nc*) anders dan een persoonlijk voornaamwoord is altijd *3e persoon*, vandaar dat voor de persoon al een *3* in de kenmerkenlijst van de betreffende DCG-regel is ingevuld. Voor wat betreft de kenmerken van een *vc* zij men verwezen naar opgave ***4.4***.

```
zin([])-->
        nc([P,G]),
        vc([P,G]).

nc([P,G])-->pvnw([P,G]).
nc([3,G])-->znw([G]).                    /* Dit type nc heeft P=3 ! */

vc([P,enkelvoud])-->
        kww([P,enkelvoud]),
        nc([_,enkelvoud]).
vc([P,meervoud])-->
        kww([P,meervoud]),
        nc([_,_]).

pvnw([1,enkelvoud])-->[ik].
pvnw([3,enkelvoud])-->[zij].

kww([1,enkelvoud])-->[ben].
kww([3,enkelvoud])-->[is].

znw([enkelvoud])-->[programmeur].
```

De vertaling van de tweede versie van onze grammatica ziet er als volgt uit:

```
zin([],A,B) :-
        nc([P,G],A,C),
        vc([P,G],C,B).

nc([P,G],A,B) :-
        pvnw([P,G],A,B).
nc([3,G],A,B) :-
        znw([G],A,B).

vc([P,enkelvoud],A,B) :-
        kww([P,enkelvoud],A,C),
        nc([_,enkelvoud],C,B).
vc([P,meervoud],A,B) :-
        kww([P,meervoud],A,C),
        nc([_,_],C,B).
pvnw([1,enkelvoud],[ik|B],B).
pvnw([3,enkelvoud],[zij|B],B).

kww([1,enkelvoud],[ben|B],B).
kww([3,enkelvoud],[is|B],B).

znw([enkelvoud],[programmeur|B],B).
```

—Argumenten van nonterminals in de grammatica worden letterlijk in de vertaling opgenomen en wel *vóór* de twee extra argumenten die de zin en het overblijvende deel bevatten.

De werking van de vertaalde DCG is gelijk aan die van de vorige versie, behalve dan dat bij het ontleden/genereren rekening gehouden wordt met de unificeerbaarheid van de extra argumenten van de diverse conjuncten. Uiteraard moeten we er bij het gebruik van de predikaten rekening

mee houden dat er vóór de twee argumenten die de woordlijsten bevatten een argument bijgekomen is, waardoor alle predikaten *drietallig* zijn:

```
| ?-zin([ik,ben,programmeur],[]).
no
| ?-zin([],[ik,ben,programmeur],[]).
yes
| ?-zin(K,[ik,is,programmeur],[]).
no
| ?-nc(K,[programmeur,prolog],Rest).
K=[3,enkelvoud]
Rest=[prolog]
yes
| ?-nc([1,enkelvoud],[zij],[]).
no
| ?-nc([3,enkelvoud],[zij],_).
yes
| ?-zin([],[programmeur,Werkwoord,ik],[]).
Werkwoord=is;
no                                        /* ben is dus géén oplossing */
```

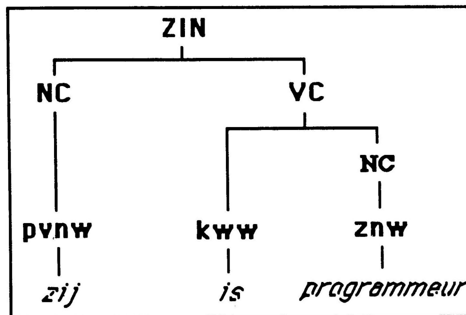
Zinnen zoals in de laatste vraag zullen we later door middel van semantische kenmerken uitsluiten, waarbij we er tevens zorg voor zullen dragen dat het correcte zinnetje *programmeur ben ik* wél tot de verzameling van correcte zinnen gaat behoren.

11.3.1 Opgave

11.2 Rust de grammatica van opgave ***11.1*** uit met de nodige kenmerken en experimenteer ermee (na een copie in *gramma2.pro* gemaakt te hebben).

11.4 Parser 1: regeltoepassing weergevende structuren

Grammatici zijn niet gelukkig voordat ze aan een zin een boomstructuur hebben toegekend. In het eenvoudigste geval correspondeert een ontleedboom precies met de manier waarop de herschrijfregels zijn toegepast: iedere vertakking is het gevolg van de toepassing van een regel. Bekijken we bijvoorbeeld de boom uit 4.3.3 nog eens:



In deze boom is de hoogste vertakking ontstaan door toepassing van de regel **zin-->nc,** vc; de

vc is ontstaan door toepassing van de regel **vc-->kww,znw**; enzovoorts. De boomstructuur kunnen we, zoals we in hoofdstuk 4 hebben gezien, weergeven met de volgende haakjesnotatie:

zin(nc(pvnw(zij)),vc(kww(is),znw(programmeur)))

Om de weergave van een *zin* in een dergelijke structuur te kunnen produceren, maken we opnieuw gebruik van de mogelijkheid nonterminals van argumenten te voorzien. We voegen aan elke nonterminal een argument toe, dat de structuur bevat die door die nonterminal opgeleverd wordt. Hierbij is er een verschil tussen de nonterminals aan de linkerkant van een herschrijfregel en die aan de rechterkant. In de nonterminals aan de *linkerkant* wordt de structuur *expliciet* weergegeven, en daarmee gedefinieerd; de nonterminals aan de *rechterkant* bevatten slechts een *variabele* die tijdens een ontleding geünificeerd moet worden met de structuur die die nonterminal oplevert. Deze variabelen worden ook gebruikt om de structuur in de nonterminal aan de linkerkant te definiëren. Aan de hand van twee voorbeelden zullen we zien hoe de regels uitgebreid moeten worden. Eerst een voorbeeld van een *lexicale regel*:

znw([enkelvoud])-->[programmeur].

We beginnen met het reserveren van ruimte voor een extra argument in de enige nonterminal:

znw([enkelvoud],...............)-->[programmeur].

De **znw** nonterminal moet een *znw-knoop* construeren, met andere woorden ***znw*** is de functor van de te bouwen structuur.

znw([enkelvoud],znw(..........))-->[programmeur].

Volgens de regel bestaat een *znw* uit de terminal *programmeur*. De tak ***znw*** staat hiërarchisch dus direct boven de terminal. Dit geven we weer door de eindterm als argument van de functor mee te geven, waarmee de nieuwe versie van de regel voltooid is:

znw([enkelvoud],znw(programmeur))-->[programmeur].

Toepassingen van deze regel na vertaling zijn bijvoorbeeld:

```
| ?-znw(K,ZNW,[programmeur],[]).
K=enk
ZNW=znw(programmeur)
yes
| ?-znw(K,ZNW,[programmeur,teveel],[]).
no
```

Een voorbeeld van een regel met *nonterminals aan de rechterkant* is:

zin([])-->nc([P,G]),   vc([P,G]).

We beginnen weer met het reserveren van ruimte voor een extra argument in de nonterminals:

zin([],..........)-->nc([P,G],...),   vc([P,G],...).

De nonterminal *zin* moet een *zin*sstructuur retourneren, dus de functor van de te retourneren structuur is ***zin***:

```
zin([],zin(.....))-->nc([P,G],....), vc([P,G],....).
```

Elke nonterminal aan de rechterkant levert bij adequate input een structuur op die correspondeert met een deel van de zin. De variabele waarmee de structuur geünificeerd wordt, geven we een naam die sprekend lijkt op de naam van de nonterminal, namelijk de nonterminal in hoofdletters (*NC* en *VC*). Vervolgens zetten we deze namen op de juiste plaats:

```
zin([],zin(.....))-->nc([P,G],NC), vc([P,G],VC).
```

Volgens de regel bestaat een *zin* uit een *nc* en een *vc*. De structuren hiervan zijn geünificeerd met respectievelijk de variabelen NC en VC. Deze moeten dan ook als argumenten meegegeven worden aan de functor ***zin*** in het hoofd van de regel, om de zinsstructuur te kunnen beschrijven:

```
zin([],zin(NC,VC))-->nc([P,G],NC), vc([P,G],VC).
```

Wanneer we alle regels van onze grammatica op deze wijze uitbreiden, hebben we van onze *acceptor* een *ontleder* gemaakt. De derde versie van onze DCG luidt:

```
zin([],zin(NC,VC))-->
        nc([P,G],NC),
        vc([P,G],VC).

nc([P,G],nc(PVNW))-->pvnw([P,G],PVNW).
nc([3,G],nc(ZNW))-->znw([G],ZNW).

vc([P,enkelvoud],vc(KWW,NC))-->
        kww([P,enkelvoud],KWW),
        nc([_,enkelvoud],NC).
vc([P,meervoud],vc(KWW,NC))-->
        kww([P,meervoud],KWW),
        nc([_,_],NC).

pvnw([1,enk],pvnw(ik))-->[ik].
pvnw([3,enk],pvnw(zij))-->[zij].

kww([1,enk],kww(ben))-->[ben].
kww([3,enk],kww(is))-->[is].

znw([enk],znw(programmeur))-->[programmeur].
```

Na vertaling kunnen we het predikaat zin/4 bijvoorbeeld als volgt gebruiken:

```
| ?-zin(_,Boom,[zij,is,programmeur],[]).
Boom=zin(nc(pvnw(zij)),vc(kww(is),znw(programmeur)))
yes
| ?-zin(_,B,[Wie,is,programmeur],[]).
Wie=zij
B=zin(nc(pvnw(zij)),vc(kww(is),nc(znw(programmeur))));
Wie=programmeur
B=zin(nc(znw(programmeur)),vc(kww(is),nc(znw(programmeur))))
yes
```

We kunnen ook zinnen met een voorgeschreven structuur genereren door de gewenste structuur te specificeren in het tweede argument. De volgende vraag bijvoorbeeld produceert zinnen met als eerste woord een persoonlijk voornaamwoord en als derde een zelfstandig naamwoord:

```
| ?-zin([],zin(nc(pvnw(P)),vc(V,nc(znw(Z)))),Zin,[]).
P=ik,
V=kww(ben),
Z=programmeur,
Zin=[ik,ben,programmeur];
P=zij,
V=kww(is),
Z=programmeur,
Zin=[zij,is,programmeur];
no
```

Vertalen we de derde versie van onze DCG, dan krijgen we:

```
zin([],zin(NC,VC),A,B):-
        nc([P,G],NC,A,C),
        vc([P,G],VC,C,B).

nc([P,G],nc(PVNW),A,B):-
        pvnw([P,G],PVNW,A,B).
nc([3,G],nc(ZNW),A,B):-
        znw([G],ZNW,A,B).

vc([P,enkelvoud],vc(KWW,NC),A,B):-
        kww([P,enkelvoud],KWW,A,C),
        nc([_,enkelvoud],NC,C,B).
vc([P,meervoud],vc(KWW,NC),A,B):-
        kww([P,meervoud],KWW,A,C),
        nc([_,_],NC,C,B).

pvnw([1,enkelvoud],pvnw(ik),[ik|B],B).
pvnw([3,enkelvoud],pvnw(zij),[zij|B],B).

kww([1,enkelvoud],kww(ben),[ben|B],B).
kww([3,enkelvoud],kww(is),[is|B],B).

znw([enkelvoud],znw(programmeur),[programmeur|B],B).
```

—Het belangrijkste dat vermeld moet worden over de werking van de derde versie van onze DCG als programma is dat er gewerkt wordt met *incomplete* boomstructuren, omdat gedurende het ontleedproces delen van de boom nog slechts met een variabele worden aangeduid. Pas wanneer een woord herkend is, wordt een deelstructuur helemaal vastgelegd.
—Een andere aantekening die moet worden gemaakt, heeft betrekking op de *naamgeving van de structuurvariabelen*: het is heel wel mogelijk dat een regel dezelfde nonterminal meer dan eens bevat, zónder dat de door die nonterminal opgeleverde structuren gelijk behoeven te zijn. In dat geval moeten de variabelen in de verschillende nonterminals ongelijk zijn aan elkaar. Bijvoorbeeld: de *vc* met twee *nc*'s in de zin *Jan geeft marie een boek* kan beschreven worden door de volgende grammaticale regel (de syntactische kenmerken zijn weggelaten):

```
vc(vc(WW,NC_1,NC_2))-->ww(WW),nc(NC_1),nc(NC_2).
```

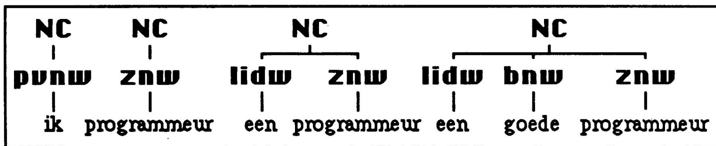
11.4.1 Opgaven

11.3 Definieer een predikaat **pkz/0** om alle zinnen van het type *pvnw+kww+znw* die door de derde grammatica beschreven worden op het scherm te tonen:

```
| ?-pkz.
[ik,ben,programmeur]
[zij,is,programmeur]

yes
```

11.4 Naamwoordconstituenten kunnen behalve de al getoonde structuren uiteraard ook andere structuren hebben, zoals de twee meest rechtse in het volgende plaatje:



(a) Breid de derde versie van de grammatica uit zodat ook deze structuren het resultaat van een ontleding kunnen zijn:

```
| ?-nc(NC,[een,programmeur],[]).
NC=nc(lidw(een),znw(programmeur))
yes
| ?-nc(_,[W1,W2,W3],[]).
W1=een,
W2=goede,
W3=programmeur
yes
```

Voor de nieuwe structuren in het plaatje moet het woordenboek worden uitgebreid met enige bijvoeglijke naamwoorden. Bijvoeglijke naamwoorden (***bnw***) kunnen in een *nc* twee vormen hebben, nl. met en zonder een "*e*" op het eind. Welke vorm gebruikt moet worden hangt af van het lidwoord dat *bepaald* (de of het) of *onbepaald* (een) kan zijn en van het zelfstandig naamwoord dat *de* of *het* als lidwoord verlangt. Voeg aan de definities in het lexicon de nodige kenmerken toe en ook enige zelfstandige naamwoorden , zodat uiteindelijk minstens de volgende sessie mogelijk is:

```
| ?-nc(NC,[een,programmeur],[]).
NC=nc(lidw(een),znw(programmeur))
yes
| ?-nc(_,[W1,W2,W3],[]).
W1=een,
W2=goede,
W3=programmeur
yes

| ?-zin(Z,[ik,ben,een,goed,mens],[]).
Z=zin(pvnw(ik),kww(ben),nc(lidw(een),bnw(goed),znw(mens))).
yes
```

```
| ?-zin(Z,[hij,is,een,goede,programmeur],[]).
Z=zin(pvnw(hij),kww(is),nc(lidw(een),bnw(goede),znw(programmeur))).
yes
```

(b) Schrijf een programma **pp/1** om structuren netjes naar het scherm te schrijven. Een mogelijke lay-out is die van bladzijde 52.

(c) Bouw de grammatica verder uit (onder meer met ***voorzetsels*** en voorzetselconstituenten (*vzc*'s)) zodat ook zinnen als de volgende ontleed kunnen worden:

Hij koopt een goede computer.
Hij koopt een computer met kleurenbeeldscherm.
Hij gaf jan een dik boek.

Is hij programmeur?
Zag hij het meisje met de verrekijker? (*ambigu!*)

11.5 Parser 2: logische structuren (condities)

In de vorige versie van onze grammatica bestond er een eenvoudig verband tussen een regel en de door die regel afgeleverde structuur: de regel werd weerspiegeld in het structuurargument van het hoofd van die regel. Het verband tussen de regel en de structuur is zo strict, dat de structuurargumenten automatisch door een vertaler zouden kunnen worden meegeleverd. Aan elke nonterminal worden in dat geval dus drie, in plaats van twee argumenten toegevoegd. Een dergelijke vertaler zou de tweede versie van onze grammatica vertalen in de (normale) vertaling van de derde versie, hetgeen het definiëren van een grammatica zou vereenvoudigen.

Het verband tussen de grammaticale regels en de opgeleverde structuur is vaak minder eenvoudig als de structuur de betekenis van een constituent moet weergeven. In onderstaande versie van onze grammatica construeren we op basis van een aangeboden zinnetje een propositie die de betekenis van dat zinnetje representeert:

```
:-current_op(P,A,mod),op(P,A,:).

zin(prop:Sem,[])-->
      nc(_:Ind,[P,G]),  vc(prop:Sem0,[P,G]),
      {Sem0 =.. [Pred,Eig,Ind],
      vereenvoudig(Sem0,Sem)},
      [.].

nc(persoon:Ind,[P,G])-->pvnw(persoon:Ind,[P,G]).
nc(klasse:Klasse,[3,G])-->znw(klasse:Klasse,[G]).

vc(prop:is_een(Klasse,_),[P,G])-->
      kww(prop:is_een(Klasse,_),[P,G]),
      nc(klasse:Klasse,_).

/* Lexicon */
pvnw(persoon:ik,[1,enkelvoud])-->[ik].
pvnw(persoon:zij,[3,enkelvoud])-->[zij].

kww(prop:is_een(Klasse,Ind),[1,enkelvoud])-->[ben].
```

```
kww(prop:is_een(Klasse,Ind),[3,enkelvoud])-->[is].

znw(klasse:programmeur,[enkelvoud]) -->[programmeur].

/* Hulp predikaat */
vereenvoudig(Sem0,Sem):-
        Sem0=..[is_een,Prop,Ind],!,
        (
                Prop=Ind,!,
                Sem=true
        ;
                Sem=..[Prop,Ind]
        ).
vereenvoudig(Sem,Sem):-
        Sem=..Sem0.
```

Om te laten zien wat de bedoeling is, genereren we alle drie mogelijke zinnen en de bijbehorende propositie:

```
| ?-zin(Clausule,_,Zin,[]).
Clausule=programmeur(ik),
Zin=[ik,ben,programmeur,.];
Clausule=programmeur(zij),
Zin=[zij,is,programmeur,.];
Clausule=true,
Zin=[programmeur,is,programmeur,.];
no
```

Dit lijkt misschien op het baren van drie muizen door een olifant, maar alleen al door de toevoeging van een paar regels kan de productiviteit van deze grammatica explosief toenemen, waardoor deze toch een zuinige representatie voor een groot aantal zinnen is. Vertaling van de grammatica levert de volgende parser op:

```
:-current_op(P_91,A_91,mod),op(P_91,A_91,:).
zin(prop:Sem,[],A,B):-
        nc(_:Ind,[P,G],A,C),
        vc(prop:Sem0,[P,G],C,[.|B]),
        Sem0=..[Pred,Eig,Ind],
        vereenvoudig(Sem0,Sem).

nc(persoon:Ind,[P,G],A,B):-
        pvnw(persoon:Ind,[P,G],A,B).
nc(klasse:Klasse,[3,G],A,B):-
        znw(klasse:Klasse,[G],A,B).

vc(prop:is_een(Klasse,_),[P,G],A,B):-
        kww(prop:is_een(Klasse,_),[P,G],A,C),
        nc(klasse:Klasse,_,C,B).

pvnw(persoon:ik,[1,enkelvoud],[ik|B],B).
pvnw(persoon:zij,[3,enkelvoud],[zij|B],B).

kww(prop:is_een(Klasse,Ind),[1,enkelvoud],[ben|B],B).
```

```
kww(prop:is_een(Klasse,Ind),[3,enkelvoud],[is|B],B).

znw(klasse:programmeur,[enkelvoud],[programmeur|B],B).

vereenvoudig(Sem0,Sem):-
      Sem0=..[is_een,Eig,Ind],
      !,
      (
            Eig=Ind,!,
            Sem=true
      ;
            Sem=..[Eig,Ind]
      ).
vereenvoudig(Sem0,Sem):-
      Sem=..Sem0.
```

We zien dat de vertaler *normale clausules* (zoals de definitie van het predikaat **vereenvoudig**/2) en *directieven* (zoals dat waarin de dubbele punt (:) als operator gedeclareerd wordt) ongemoeid laat. Ook zien we hoe grammaticale condities (zoals die in de herschrijfregel voor **zin**) onveranderd opgenomen worden in de clausule die het resultaat is van de vertaling.

Maar nu het het "semantische gedeelte". Iedere nonterminal bevat behalve de syntactische kenmerken ook een argument van de vorm:

Type : Structuur

Door nu in de grammatica de typen zorgvuldig met elkaar te combineren kunnen de juiste structuren gebouwd worden:
- Een **nc** levert een structuur van het type ***klasse*** op.
- Een **pvnw** levert een structuur van het type ***persoon*** op.
- Een vc levert via het koppelwerkwoord een *raamwerk voor een tweeplaatsige propositie met het predikaat* ***is_een*** op. Het eerste argument (***Eig***) van de propositie wordt in de *vc* regel geünificeerd met de ***klasse*** in de *nc* van diezelfde regel.
- Het raamwerk wordt in de herschrijfregel voor de nonterminal **zin** omgezet in een lijstje van drie elementen (door middel van de *univ* operator), waarbij het tweede nog variabele argument van het raamwerk (derde element in het lijstje!), geünificeerd wordt met de door **nc** opgeleverde individu ***Ind***: de regel voor *zin* is

```
zin(prop:Sem,[])-->
      nc(_:Ind,[P,G]),
      vc(prop:Sem0,[P,G]),
      {
            Sem0 =.. [Pred,Eig,Ind],
            vereenvoudig(Sem0,Sem)
      },
      [.].
```

Als de conditie aan de beurt is, geldt in geval van het zinnetje *[zij,is,programmeur]* de volgende substitutie:

Sem0=is_een(programmeur,Var), Ind=zij

De *univ* operator in de conditie leidt tot de volgende uniVicatie:

is_een(programmeur,Var) =.. [is_een,programmeur,zij]

hetgeen leidt tot de substitutie

Var=zij, Sem0=is_een(programmeur,zij).

vereenvoudig/2 levert tenslotte door de reductie van ***Sem0*** op:

Sem=programmeur(zij).

Een waarheid als "programmeur is programmeur" reduceren we tot **true**.

11.5.1 Opgaven

11.5 Wijzig de vierde versie van de grammatica, zodat zinnetjes van het type

jan is goed

omgezet worden in een propositie van het type:

heeft_eig(jan,goed)

11.6 Combineer parsers 1 en 2 tot een parser die zowel een syntactische als een logische structuur teruggeeft.

11.6 Andere grammaticaformalismen

Het DCG formalisme (Pereira en Warren 1980), gebaseerd op de Metamorphosis Grammar van Colmerauer (1978) is het begin geweest van een ontwikkeling die nog steeds aan de gang is. Er wordt met allerlei andere formalismen geëxperimenteerd. De belangrijkste zijn Extraposition Grammar (Pereira), Definite Clause Translation Grammars (Abramson), Gapping Grammars (Dahl & Abramson), Modular Logic Grammars (McCord), Restriction Grammars (Hirschman & Puder), Meta Restriction Grammars (Hirschman). Voor een introductie zie Dahl en Saint-Dizier (1985) en Dahl en Abramson (1987).

Literatuur

Bratko, I. (1986), *Prolog Programming for Artificial Intelligence.* Addison-Wesley.

Clocksin, W.F. en C.S. Mellish (1981), *Programming in Prolog.* Springer-Verlag, Berlijn (2nd ed.), 1984.

Coelho, H., J.C. Cotta en L.M. Pereira (1980), *How to solve it with Prolog.* Laboratório Nacional de Engenharia Civil, 4e editie, Lissabon 1985.

Dahl, V. en P. Saint-Dizier (eds) (1985), *Natural Language Understanding and Logic Programming.* Elsevier-Science Publishers.

Dahl, V. en H. Abramson (1987), *Logic Grammars.* Springer Verlag, Berlijn.

Dennett, D.C. (1978), *Brainstorms. Philosophical Essays on Mind and Psychology.* Bradford Books, Montgomery.

Fodor, J.A. (1983), *The Modularity of Mind.* The MIT Press, Cambridge (MA).

Gabriel, J., T. Lindholm, E.L. Lusk en R.A. Overbeek (1986), *A Tutorial on the Warren Abstract Machine for Computational Logic.* Mathematics and Computer Science Division, Argonne National Laboratory, Argonne (Ill).

Giannesini, F., H. Kanoui, R. Pasero en M. van Caneghem (1986), *Prolog.* Addison-Wesley.

Goos G. en J. Harmanis (Eds.) (1986), *Proceedings of the Third International Conference on Logic Programming.* Springer Verlag, Berlijn.

Hagoort, P. en R. Maessen (Red.) (1986) *Geest, computer, kunst.* Stichting Grafiet, Utrecht.

Hofstadter, D.R. (1979), *Gödel, Escher, Bach: an Eternal Golden Braid.* Vintage Books, New York, 1980.

Kluzniak, F. en S. Szpakowicz (1985), *Prolog for Programmers.* Academic Press, London.

Kowalski, R. (1979), Algorithm = Logic+Control. *Communications of the ACM,* 22, 424-436.

LLoyd, J.W. (1984), *Foundations of Logic Programming.* Springer Verlag, Berlijn.

Moss, C. (1986), Cut & Paste - Defining the Impure Primitives of Prolog. In: G. Goos en J. Harmanis (Eds.), *Proceedings of the Third International Conference on Logic Programming.* Springer Verlag, Berlijn.

Nagel, E. en J.R. Newman (1958), *De stelling van Gödel.* Het Spectrum, Utrecht, 1975.

Pereira, F.C.N. en D.H.D. Warren (1980), Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence, 13,* 231-278.

Pylyshyn, Z.W. (1984), *Computation and Cognition. Toward a Foundation for Cognitive Sciences.* The MIT Press, Cambridge (MA).

Ringwood, G.A. (1986), *The Dining Logicians.* Imperial College, Department of Computing, London.

Roos, N. (1987), *Evaluatie van Prolog Implementaties.* TU Delft, Faculteit der Wiskunde en Informatica, Vakgroep Informatica.

Schank, R.C. en R.P. Abelson (1977), *Scripts, Plans, Goals and Understanding.* Erlbaum, Hillsdale (NJ).

Seuren, P.A.M (1985), *Discourse Semantics.* Basil Blackwell, Oxford.

Sterling, L. en E. Shapiro (1986), *The art of Prolog. Advanced Programming Techniques.* The MIT Press, Cambridge (MA).

Warren, D.H.D. (1983), *An Abstract Prolog Instruction Set.* Technical Note 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International.

OPLOSSINGEN

```
                                        /* 2.5 (a) */
dier(X):-hond(X).
dier(X):-poes(X).
                                        /* 2.5 (b) */
sterfelijk(X):- dier(X).
```

2.5 (d) De dierklasse die als eerste gedefinieerd staat levert ook als eerste oplossingen. Staan de klassen zoals hierboven in (a), dan komen eerst de honden en daarna de poezen. Worden deze twee regels omgekeerd, dan komen eerst de poezen en daarna de honden.

2.5 (e) consult

```
                                        /* 3.1 (b) */
zinnetje(P,W,Z):-
        persoonlijk_voornaamwoord(P,Persoon,Getal),
        koppelwerkwoord(W,Persoon,Getal),
        zelfstandig_naamwoord(Z,Getal).
                                        /* 3.1 (c) */
vraagje(W,P,Z):-
        koppelwerkwoord(W,Persoon,Getal),
        persoonlijk_voornaamwoord(P,Persoon,Getal),
        zelfstandig_naamwoord(Z,Getal).
                                        /* 3.1 (d) */
zinnetje_vraagje(P,W,Z,W,P,Z):-
        zinnetje(P,W,Z),
        vraagje(W,P,Z).
                                          /* OF */
zinnetje_vraagje(P,W,Z,W,P,Z):-
        koppelwerkwoord(W,Persoon,Getal),
        persoonlijk_voornaamwoord(P,Persoon,Getal),
        zelfstandig_naamwoord(Z,Getal).
                    /* 3.2 Alleen de mannelijke relatietermen zijn opgenomen:*/
zoon(Z,O):- kind(Z,O),is_man(Z).
ouder(O,K):- kind(K,O).
vader(V,K):- ouder(V,K),is_man(V).
grootouder(G,K):- ouder(G,X), ouder(X,K).
grootvader(G,K):- vader(G,X), ouder(X,K).
kleinkind(K,G):- grootouder(G,K).
kleinzoon(K,G):- grootouder(G,K), is_man(K).
/* Het probleem met onderstaande definitie is o.a. dat een man broer van zichzelf is! */
broer(A,B):- ouders(V,M,A), ouders(V,M,B), is_man(A).
                                          /* 4.2 */
nc(woorden(Wrd),nc(znw(Wrd))):-
        zelfstandig_naamwoord(Wrd,_).
nc(woorden(Wrd),nc(pvnw(Wrd))):-
        persoonlijk_voornaamwoord(Wrd,_,_).
                                          /* 4.3 */
nc(woorden(Wrd),nc(znw(Wrd)),kenmerken(3,Getal)):-
        zelfstandig_naamwoord(Wrd,Getal).
nc(woorden(Wrd),nc(pvnw(Wrd)),kenmerken(Persoon,Getal)):-
        persoonlijk_voornaamwoord(Wrd,Persoon,Getal).
```

```
                                        /* 4.4 */
vc(woorden(W1,W2),vc(kww(W1),NC),kenmerken(Persoon,enkelvoud)):-
        koppelwerkwoord(W1,Persoon,enkelvoud),
        nc(woorden(W2),NC,kenmerken(_,enkelvoud)).
vc(woorden(W1,W2),vc(kww(W1),NC),kenmerken(Persoon,meervoud)):-
        koppelwerkwoord(W1,Persoon,meervoud),
        nc(woorden(W2),NC,kenmerken(_,_)).
zin(woorden(W1,W2,W3),zin(nc(pvnw(W1)),VC)):-
        nc(woorden(W1),nc(pvnw(W1)),kenmerken(P,G)),
        vc(woorden(W2,W3),VC,kenmerken(P,G)).
| ?-zin(W,Structuur).
W       = woorden(ik,ben,programmeur),
Structuur       = zin(nc(pvnw(ik)),vc(kww(ben),nc(znw(programmeur))));
W       = woorden(ik,ben,psycholoog),
Structuur       = zin(nc(pvnw(ik)),vc(kww(ben),nc(znw(psycholoog))));
W       = woorden(ik,ben,ik),
Structuur       = zin(nc(pvnw(ik)),vc(kww(ben),nc(pvnw(ik))))
yes
| ?-zin(W,Structuur).
W       = woorden(ik,ben,programmeur),
Structuur       = zin(nc(pvnw(ik)),vc(kww(ben),nc(znw(programmeur))));
W       = woorden(ik,ben,psycholoog),
Structuur       = zin(nc(pvnw(ik)),vc(kww(ben),nc(znw(psycholoog))));
W       = woorden(ik,ben,ik),
Structuur       = zin(nc(pvnw(ik)),vc(kww(ben),nc(pvnw(ik))));
W       = woorden(ik,ben,jij),
Structuur       = zin(nc(pvnw(ik)),vc(kww(ben),nc(pvnw(jij))));
W       = woorden(ik,ben,u),
Structuur       = zin(nc(pvnw(ik)),vc(kww(ben),nc(pvnw(u))));
W       = woorden(ik,ben,hij),
Structuur       = zin(nc(pvnw(ik)),vc(kww(ben),nc(pvnw(hij))));
W       = woorden(ik,ben,zij),
Structuur       = zin(nc(pvnw(ik)),vc(kww(ben),nc(pvnw(zij))));
W       = woorden(jij,bent,programmeur),
Structuur       = zin(nc(pvnw(jij)),vc(kww(bent),nc(znw(programmeur))));
Enzovoorts
/* 4.5: Geen lijst zijn de formules 1,3,5,8,9,10,12,13 */
                                        /* 4.6 (Gedeeltelijk) */
zin([W1,W2,W3],zin(nc(pvnw(W1)),VC)):-
        nc([W1],nc(pvnw(W1)),[P,G]),
        vc([W2,W3],VC,[P,G]).
                                        /* 4.8 (Gedeeltelijk) */
kop([K|_],K).
staart([_|S],S).
staart_van_kop([[_|S]|_],S).  /* De kop moet zelf wel een lijst zijn! */
                                        /* 5.3 */
laatste([X],X).
laatste([E1,E2|S],X):-laatste([E2|S],X).
                                        /* 5.5 (a) */
voor(X,Y,[X|S]):-is_elem_van(Y,S).
voor(X,Y,[_|S]):-voor(X,Y,S).
                                        /* 5.5 (b) */
na(X,Y,L):-voor(Y,X,L).
```

```
                                                            /* 5.5 (c) */
kaart(K,W):-
        kleur(K),
        waarde(W).
kleur(schoppen).
kleur(harten).
kleur(ruiten).
kleur(klaveren).
waarde(aas).
waarde(etcera).
hogere_kleur(kaart(K1,_),kaart(K2,_)):-
        voor(K1,K2,[schoppen,harten,ruiten,klaveren]).
                                                            /* 5.6 */
pick(Lijst,El,Overige):-
        append(L,[El|S],Lijst),
        append(L,S,Overige).
                                                            /* 5.7 */
permutatie([],[]).
permutatie(Lijst,[Element|Permutatie]):-
        pick(Lijst,Element,Overige),
        permutatie(Overige,Permutatie).
/* 5.8 Zie tekst van opgave 10.6: Laat de clausules die voorzien zijn van het commentaar /* nieuwe clausule */
weg en een groot deel van de oplossing is verkregen. */
                                                            /* 5.10 */
knopen(leeg,[]).
knopen(boom(Knoop,L,R),Knopen):-
        knopen(L,L_knopen),
        knopen(R,R_knopen),
        append([Knoop|L_knopen],R_knopen,Knopen).
/* 6.1 Genereren van elementen gaat niet meer ! */
                                                            /* 6.2 */
na_elem(X,[X|S],S):-!.
na_elem(X,[_|S],L):-
        na_elem(X,S,L).

                                                            /* 6.3 (a) */
maak_verzameling([],[]).
maak_verzameling([K|S],[K|Verz]):-
        schrap(K,S,Geschoond),
        maak_verzameling(Geschoond,Verz).
                                                            /* 6.3 (b) */
deelverzameling([],_).
deelverzameling([E|S],V):-
        is_elem_van(E,V),
        deelverzameling(S,V).
                                                            /* 6.3 (c) */
identiek1(V,W):-
        deelverzameling(V,W),
        deelverzameling(W,V).
identiek2([],[]).
identiek2([E|S],V):-
        pick(V,E,W),
        identiek2(S,W).
```

```
                                        /* 6.3 (d) */
doorsnede([],_,[]).
doorsnede([E|S1],V2,[E|D3]):-!,
        is_elem_van(E,V2),
        doorsnede(S1,V2,D3).
doorsnede([_|S1],V2,D3):-
        doorsnede(S1,V2,D3).
/* 6.3 -> e,f,g,h zelf doen! */
                    /* 6.4 (b) {(a) ontstaat uit (b) door 3e en 4e clausule weg te laten} */
vervang_n(_,_,[],[]).
vervang_n(X,Y,[X|S_in],[Y|S_uit]):-!,
        vervang_n(X,Y,S_in,S_uit).
vervang_n(X,Y,[[X|S_in]|T_in],[[Y|S_uit]|T_uit]):-!,
        vervang_n(X,Y,S_in,S_uit),
        vervang_n(X,Y,T_in,T_uit).
vervang_n(X,Y,[[Z|S_in]|T_in],[[Z|S_uit]|T_uit]):-!,
        vervang_n(X,Y,S_in,S_uit),
        vervang_n(X,Y,T_in,T_uit).
vervang_n(X,Y,[Z|S_in],[Z|S_uit]):-
        vervang_n(X,Y,S_in,S_uit).
                                        /* 6.5 */
plet([[K|S]|T],P):-!,
        plet([K|S],Pks),
        plet(T,Pt),
        append(Pks,Pt,P).
plet([[]|S],Ps):-!,
        plet(S,Ps).
plet([K|S],[K|Ps]):-!,
        plet(S,Ps).
plet([],[]):-!.
                                        /* 6.6 */
klinker(Letter):-
        is_elem_van(Letter,[a,e,i,o,u,y]).
medeklinker(y):- !.
medeklinker(Letter):-
        klinker(Letter), !, fail.
medeklinker(_).
                                        /* 6.7 */
is_geen_element_van(E,L):-
        is_elem_van(E,L),!,
        fail.
is_geen_element_van(_,_).
                                        /* 7.1 */
maximum([G1|Rest],Max):-
        maximum(Rest,G1,Max).
maximum([],Max,Max).
maximum([G|Rest],Voorlopig_max,Max):-
        G >= Voorlopig_max,!,
        maximum(Rest,G,Max).
maximum([G|Rest],Voorlopig_max,Max):-
        maximum(Rest,Voorlopig_max,Max).
                                        /* 7.3 */
nth([E|_],1,E).
nth([_|S],N,E):-
        N_1 is N-1,
        nth(S,N_1,E).
```

```
                                        /* 7.5 */
som(Getal1,Getal2,Som):-
        reverse(Getal1,Reversed1),
        reverse(Getal2,Reversed2),
        gpg(Reversed1,Reversed2,0,[],Som).
gpg([],[],0,Werksom,Werksom):-!.
gpg([],[],Overloop,Werksom,[Overloop|Werksom]):-!.
gpg([],Getal2,0,Werksom,Som):-
        reverse(Getal2,R),
        append(R,Werksom,Som),!.
gpg([],Getal2,Overloop,Werksom,Som):-
        gpg([Overloop],Getal2,0,Werksom,Som),!.
gpg(Getal1,[],0,Werksom,Som):-
        reverse(Getal1,Reversed1),
        append(Reversed1,Werksom,Som),!.
gpg(Getal1,[],Overloop,Werksom,Som):-
        gpg([Overloop],Getal1,0,Werksom,Som),!.
gpg([Cijfer1|Overige1],[Cijfer2|Overige2],Overloop,Werksom,Som):-
        cpc(Cijfer1,Cijfer2,Overloop,Cijfer,Overloop_nw),
        gpg(Overige1,Overige2,Overloop_nw,[Cijfer|Werksom],Som).
product(Getal1,Getal2,Product):-
        gxg(Getal1,Getal2,[],Product).
gxg([],Getal2,Product,Product):- !.
gxg([Cijfer1|T1],Getal2,Werkproduct,Product):-      /*Eerst meest linkse cijfer*/
        cxg(Cijfer1,Getal2,CXG),
        schuifop(Werkproduct,Werkproduct0),         /*Nul erachter*/
        gpg(Werkproduct0,CXG,Som),
        gxg(T1,Getal2,Som,Product),!.
/* enzovoorts */
                                        /* 8.1 */
hallo(_,0):-!.
hallo(N,P):-
        Q is P-1,
        tab(Q),
        write('"Hallo!"'),nl,
        M is N-1,
        hallo(P,M).
                                        /* 8.3 */
ja_nee(Vraag) :-
        repeat,
        write(Vraag),
        write(' [ja.|nee.] '),
        read(X),
        (X == ja, ! ; X == nee, !, fail ; fail).
                                        /* 8.4 */
is_letter(C) :- C >= 65, C =< 90, !.
is_letter(C) :- C >= 97, C =< 122, !.
                                        /* 8.5 */
ascii(Van,Tot):-
        Van < Tot,
        oplopend(Van,Tot),!.
ascii(Van,Tot):-
        afnemend(Van,Tot).
oplopend(Van,Tot):- Van>Tot,!.
oplopend(Van,Tot):-
        write(Van),
        put(32),
        put(Van),
```

```
        nl,
        V is Van+1,
        oplopend(V,Tot).
afnemend(Van,Tot):- Van<Tot,!.
afnemend(Van,Tot):-
        write(Van),
        put(32),
        put(Van),
        nl,
        V is Van-1,
        afnemend(V,Tot).
                                                    /* 8.6 (a) */
/* Enige ASCII codes: A=65, Z=90, a=97, z=122 */
teken_ascii(Teken,Ascii):-
        name(Teken,[Ascii]).
kl_letter(C):- C>=97,C=<122.
hfd_letter(C):- C>=65,C=<90.
mk_kl_letter(C,C):- kl_letter(C),!.
mk_kl_letter(C,D):- hfd_letter(C), D is C + 32.
schuif(N,Teken,Verschoven_Teken):-
        teken_ascii(Teken,Ascii),
        mk_kl_letter(Ascii,Ascii_kl),!,
        schuif_ascii_kl(N,Ascii_kl,Verschoven_Ascii),
        teken_ascii(Verschoven_Teken,Verschoven_Ascii).
schuif(_,Teken,Teken).
                                                    /* 8.6 (b) */
schuif_ascii_kl(N,In,Uit):-
        Uit is 97 + (((In - 97) + N) mod 26).
                                                    /* 8.6 (c) */
schuif_ascii_kl(N,In,Uit):-
        Uit is 97 + (((In - 97) + (26 + N mod 26)) mod 26).
                                                    /* 8.8 */
ott_3e(Infinitief,OTT3) :-
        name(Infinitief,Inf_l),
        reverse(Inf_l,[110,101|Rev_stam]),
        ott_3e_rev(Rev_stam,Rev_OTT3),
        reverse(Rev_OTT3,OTT3_l),
        name(OTT3,OTT3_l).
vocaal(V) :- is_elem_van(V,"eiaouy").
consonant(C) :- vocaal(C),!,fail.
consonant(_).
ott_3e_rev([C,C|Rest],[116,C|Rest]) :-          /* kk|od -> tk|od */
        consonant(C), !.
ott_3e_rev([116,V1,V2|Rest],[116,V1,V2|Rest]) :-     /* tiulf -> tiulf */
        vocaal(V1),
        vocaal(V2), !.
/* enzovoorts */
                                                    /* 8.9 */
woordenboek :-
        write('OUTPUTFILE (tussen apostrophen, afsluiten met PUNT) '),
        read(Outfile),      /* Inlezen file naam */
        repeat,
        write('----'),
        nl,
        doe_woord_definities(Outfile).
```

```
doe_woord_definities(Outfile) :-
        vraag_woord(Woord),
        (       Woord = end_of_file,
                !,
                sluit_output_file(Outfile),
                true     /* Einde cyclus ! */
        ;
                doe_zelfstandig_naamwoord(Outfile,Woord),
                fail
        ).
vraag_woord(Woord) :-
        tell(con),
        write('geef WOORD (afsluiten met PUNT) of ^Z '),
        read(Woord).
doe_zelfstandig_naamwoord(Outfile,Woord):-
        vraag_getal(Getal),
        vraag_lidwoord(Lidwoord),
        (       check_woord_definitie(Woord,Getal,Lidwoord),!,
                schrijf_woord_definitie(Outfile,Woord,Getal,Lidwoord)
        ;
                schrijf_woord_definitie_niet
        ).
vraag_getal(Getal) :-
        tab(5),
        write('GETAL'),
        nl,
        tab(10),
        (       j_n_vraag(enkelvoud,ja),
                !,
                Getal=enkelvoud
                ;
                Getal=meervoud
        ).
vraag_lidwoord(Lidwoord) :-
        tab(5),
        write('LIDWOORD'),
        nl,
        tab(10),
        (       j_n_vraag(de,ja),
                !,
                Lidwoord=de
        ;
                Lidwoord=het
        ).
check_woord_definitie(Woord,Getal,Lidwoord) :-
        write('WOORDDEFINITIE: '),
        write(zelfstandig_naamwoord(Woord,Getal,Lidwoord)),
        write('.'),
        nl,
        tab(5),
        j_n_vraag('OKEE',ja).
```

```
schrijf_woord_definitie(Outfile,Woord,Getal,Lidwoord) :-
        telling(Vorige),
        tell(Outfile),
        write(zelfstandig_naamwoord(Woord,Getal,Lidwoord)),
        write('.'),
        nl,
        tell(Vorige),
        tab(5),
        write('Woorddefinitie weggeschreven.'),
        nl.
schrijf_woord_definitie_niet:-
        tab(5),
        write('Woorddefinitie NIET weggeschreven.'),
        nl.
sluit_output_file(Outfile):-
        telling(Vorige),
        tell(Outfile),
        told,
        tell(Vorige).
                                                /* 9.3 */
vraag_kind_ouder(Prop):-
        functor(Prop,kind,2),
        (arg(1,Prop,K), var(K), !, write('Kind.  '), read(K) ; true),
        (arg(2,Prop,O), var(O), !, write('Ouder. '), read(O) ; true).
                                                /* 9.5 */
novars(X) :- var(X), !, fail.
novars(X) :- atomic(X),!.
novars([H|T]) :- !, novars(H), novars(T).
novars(Struc) :-
        Struc =.. [_|Args],
        novars(Args).
                                                /* 9.7 */
verzamel_destructief(P,[X|L]):-
        retract(P),!,
        arg(1,P,X),
        P=..[Pred,_],
        P_met_variabele =.. [Pred,Variabele],
        verzamel_destructief(P_met_variabele,L).
verzamel_destructief(_,[]).
                                                /* 9.10 */
cut(V1,V2,V3):-
        setof(X,(is_elem_van(X,V1),is_elem_van(X,V2)),V3).
symdiff(V1,V2,V3):-
        cut(V1,V2,D),
        union(V1,V2,U),
        setof(X,(is_elem_van(X,U),not(is_elem_van(X,D))),V3).
                                                /* 10.3 */
alle_operatoren :-
        current_op(Prio,Type,Oper),
        nl,
        write(Oper),tab(2),
        write(Type),tab(2),
        write(Prio),
        fail.
```

/* 10.4 */

```
:-current_op(Prioster,_,*),
        Priosqrt is Prioster-25,
        Priokwad is Priosqrt-25,
        op(Priosqrt,fy,sqrt),
        op(Priokwad,xf,kwadraat).
```

/* 10.5 */

```
:-op(400,xfx,is_een).
:-op(400,xfx,zijn).
:-op(400,xfx,heeft).
:-op(400,xfx,hebben).
:-op(300,xf,vleugels).
:-op(300,xf,poten).
/* Toe te voegen clausules: */
meervoud(twatwa,twatwas).
meervoud(vogel,vogels).
meervoud(dier,dieren).
enkelvoud(X,Y):-meervoud(Y,X).
vogels zijn dieren.
```

/* 10.6 (c) */

```
miu(In,Out):-
        match([*L,i],In),
        append(L,[i,u],Out).
miu(In,Out):-
        match([m,*X],In),
        append([m|X],X,Out).
miu(In,Out):-
        match([*X,i,i,i,*Y],In),
        append(X,[u|Y],Out).
miu(In,Out):-
        match([*X,u,u,*Y],In),
        append(X,Y,Out).
```

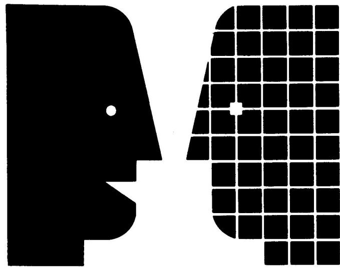
/* 10.6 (d) */

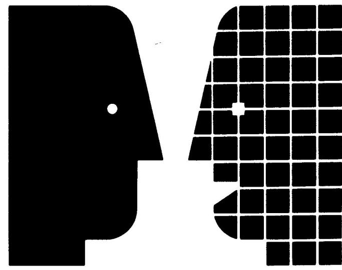
```
/* Probeer DEMO_1 en DEMO_2: : als na alle clausules van een predikaat op één na weg zijn, dan lukt assertz
+ verwijzing naar hetzelfde predikaat niet. (Zie voor meer van dit type implementatieproblemen Moss 1986) */
demo_1 :-
        abolish(predikaat,1),
        assertz(predikaat(0)),
        demo.
demo_2 :-
        abolish(predikaat,1),
        assertz(predikaat(0)),
        assertz(predikaat(1000)),
        demo.
demo :- /* Wat te doen als er echt maar 1 start fact is !!?? */
        predikaat(M),       /* Interpretator onthoudt wanneer het de laatste is !!! */
        write(predikaat(M)),nl,
        N is M+1,
        assertz(predikaat(N)),
        fail.
/* Vandaar dat begonnen wordt met minimaal 2 theorema's in de gegevensbank zitten */
miu(mi):-
        abolish(theorema,1).
miu(miu):-
/*      assert(theorema([m,i])), Indien check op dubbele theorema's gewenst is */
        assert(theorema([m,i,u])).
miu(mii):-
        assertz(theorema([m,i,i])).
```

```
miu(Nieuw):-
/*        theorema(Oud_l), Indien check op dubbele gewenst is */
          retract(theorema(Oud_l)),
          miu(Oud_l,Nieuw_l),
/*        not(theorema(Nieuw_l)),   check op dubbele theorema's*/
          assertz(theorema(Nieuw_l)),
          atoom_letters(Nieuw,Nieuw_l).      /* voor de mooiigheid */
                                     /*---------------------------------------------*/
                                     /*Alternatieve oplossing met Theo & Thea*/
                                     /*---------------------------------------------*/
init_mu:-
          abolish(theo,1),
          abolish(thea,1),
          assert(theo([m,i])).
mu(T):-
          write('theo:'),nl,
          retract(theo(Theo)),
          miu(Theo,Thea),
          assertz(thea(Thea)),
          atoom_letters(T,Thea).
mu(T):-
          write('thea:'),nl,
          retract(thea(Thea)),
          miu(Thea,Theo),
          assertz(theo(Theo)),
          atoom_letters(T,Theo).
mu(Theorema):-mu(Theorema).        /* !!!! */
atoom_letters(A,L):-
          nonvar(A),
          explode(A,L).
atoom_letters(A,L):-
          nonvar(L),
          implode(L,A).
explode(A,L):-
          name(A,Str),
          letters_string(L,Str).
implode(L,A):-
          letters_string(L,Str),
          name(A,Str).
letters_string([],[]):-!.
letters_string([L1|Lr],[C1|Cr]):-
          name(L1,[C1]),
          letters_string(Lr,Cr).
                                     /* Toegift 1: De regels zonder match */
miu(L1,L2) :-
          append(X,[i],L1) ,
          append(X,[i,u],L2).
miu([m|X],L2):-
          append([m|X],X,L2).
miu(L1,L2) :-
          append(X,[i,i,i|Y],L1) ,
          append(X,[u|Y],L2).
miu(L1,L2):-
          append(X,[u,u|Y],L1) ,
          append(X,Y,L2).
```

```
/* Toegift 2: matcher meer in de stijl van Prolog, namelijk zonder vraagteken,
           bijvoorbeeld:match([_,*V,rood,*,V],[kleur,appel,rood,x,x,[appel]]) */
:- op(100,fx,*).
match([],[]):- !.
match([V|Tp],[H|Tl]):-
        var(V),
        V=H,
        match(Tp,Tl).
match([X|Tp],L):-
        nonvar(X),
        match_nv([X|Tp],L).
match_nv([*|Tp],[_|Tl]):-
        match(Tp,Tl).
match_nv([*|Tp],[_|Tl]):-
        match([*|Tp],Tl).
match_nv([*[Hl]|Tp],[Hl|Tl]):-
        match(Tp,Tl).
match_nv([*[Hl|T]|Tp],[Hl|Tl]):-
        match([*T|Tp],Tl).
match_nv([X|Tp],[X|Tl]):-
        match(Tp,Tl).
                                        /* 10.7 */
:- current_op(Prio,_,is), op(Prio,xfx,<-).
Var <- E1+E2 :- !, Tus1 <- E1, Tus2 <- E2, gpg(Tus1,Tus2,Var).
Var <- E1*E2 :- !, Tus1 <- E1, Tus2 <- E2, gxg(Tus1,Tus2,Var).
Exp <- Exp.
                                        /* 11.3 (b) */

pp(S) :- pp(S,0).
pp( S, I ) :-
        var(S), !,
        tab(I),
        write(S),
        nl.
pp( S, I ) :-
        S =.. [ Func | Dochters ], !,
        tab( I ), write( Func ), nl,
        J is I + 3,
        ppl( Dochters, J).
pp( S, I ) :-
        tab( I ),
        write( S ),
        nl.
ppl( [], _ ) :- !.
ppl( [H|T], I ) :-
        pp( H, I ), ppl( T, I ).
```





"Who's ahead in prolog?"

"Here's looking at you, kid."

Prolog-2

Vanaf de introductie in 1984 is Prolog-2 marktleider voor serieus Prolog werk.
Prolog-2 bevat alles wat u zich kunt wensen. Het is gebouwd met onze ervaring als de eerste aanbieder van een commerciële Prolog. Al voordat de japanners besloten om Prolog in te zetten voor het 5e generatie computer projekt, was Expert Systems International op de markt met Prolog-1, een volledige Prolog implementatie waarin vele expert system shells zijn geschreven.

De nieuwe Prolog-2 FAMILIE biedt elke Prolog gebruiker een passend systeem, voor alle eisen en voor elk budget.

Prolog-2 biedt de meest uitgebreide faciliteiten, en is een complete ontwikkelomgeving voor de krachtigste computertaal die er is.

Alle tools die meegeleverd worden zijn dan ook in Prolog-2 zelf geschreven, en de source ervan is verkrijgbaar!

En Prolog-2 heeft een bijbehorende reeks expert system shells, die in Prolog-2 zelf geschreven zijn en hiermee geïntegreerd kunnen worden.

"Ambitious business-oriented Prolog development has been held back by absence of a powerful programming environment, one that Prolog-2 now provides".

PC WORLD, december 1986

- ☐ standaard Edinburgh syntax
- ☐ interpreter én compiler
- ☐ intelligent help-system
- ☐ multi-window omgeving
- ☐ menu-generator
- ☐ module system
- ☐ virtueel geheugen
- ☐ string handling
- ☐ formatted I/O
- ☐ hashed databases
- ☐ ongeveer 20K LIPS op AT
- ☐ operator syntax
- ☐ meta-programming
- ☐ definite clause grammar
- ☐ full-screen editor
- ☐ interfaces
- ☐ graphics
- ☐ voor PC, VAX-VMS en UNIX
- ☐ vanaf f 650,-

Schrijf of bel vandaag nog voor meer gegevens over onze reeks produkten.

Expert Systems International BV
Antwoordnummer 6028 -
3700 VB Zeist
Tel.: (03404)-14460
memocom (27:) NLX161

EXPERTSYSTEMS
INTERNATIONAL



Prolog is een programmeertaal die uitstekend geschikt is voor problemen, waarbij de te manipuleren gegevens complex gestructureerd zijn. Om die reden wordt Prolog hoe langer hoe meer gebruikt in het vaag afgegrensde gebied van de artificiële intelligentie, waarvan op dit moment expertsystemen de populairste toepassing zijn. Maar ook voor meer geformaliseerde vormen van symboolmanipulatie, zoals algebra, compilerbouw en computationele taalkunde is Prolog bij uitstek geschikt. Een programmeertaal die voor al deze gecompliceerde zaken geschikt is, kan uiteraard ook gebruikt worden voor meer prozaïsche programmatuur (databasesystemen, spreadsheets e.d.), zeker met de nu op de markt zijnde compilers en interpreters. Door zijn declaratieve karakter wijkt Prolog op een prettige manier af van imperatieve talen als Pascal en functionele talen als Lisp. Met Prolog kunnen al heel snel resultaten bereikt worden die in de meer conventionele talen een enorme programmeerinspanning zouden vergen. Prolog is een must voor iedereen die wat programmeertalen betreft bij wil blijven. Maar aan de andere kant is Prolog ook uitstekend geschikt als eerste programmeertaal. Door bestudering van dit boek, maar vooral door het maken van de ruim 70 opgaven, kan een ieder zich een grondige kennis van de algemeen als standaard beschouwde versie van Prolog (Edinburgh Prolog) eigen maken.

ISBN 90 6283 696 8